LINGUAGGIO C

ANSI C

Brian W. Kernighan
Dennis M. Ritchie

PREFAZIONE

Con la pubblicazione del volume *Linguaggio C*, il mondo dei calcolatori ha subito un profondo mutamento. I grandi calcolatori crescono sempre più, ed i personal computer hanno capacità paragonabili a quelle dei mainframe di una decina di anni fa. In questo periodo anche il C è cambiato, anche se di poco, e si è esteso ben oltre i limiti delle sue origini, che lo identificavano semplicemente come il linguaggio del sistema operativo UNIX.

La crescente popolarità del C, le modifiche che ha subito nel corso degli anni e la nascita di compilatori scritti da gruppi che non hanno partecipato alla sua stesura originale concorrono a dimostrare la necessità di una definizione del linguaggio più precisa ed attuale di quella fornita nella prima edizione di questo libro. Nel 1983, l'Istituto Nazionale Americano per gli Standard (ANSI) ha costituito un comitato per "una defi-nizione del linguaggio C non ambigua e non dipendente dalla macchina". Il risultato di questa ricerca è lo standard ANSI per il C.

Lo standard formalizza alcune interpretazioni suggerite, ma non descritte, nella prima edizione quali, per esempio, l'assegnamento fra strutture ed i tipi enumerativi. Esso fornisce una nuova forma di definizione della funzione, che consente il controllo incrociato della definizione stessa e delle chiamate. Specifica, inoltre, una libreria standard, con un insieme esteso di funzioni per l'input / output, la gestione della memoria, la manipolazione di stringhe ed attività affini. Lo standard specifica il comportamento di funzionalità formalizzate in modo incompleto nella prima edizione e, contemporaneamente, stabilisce esplicitamente quali aspetti del linguaggio rimangono dipendenti dalla macchina.

Questa seconda edizione de *Linguaggio C* descrive il C definito dall'ANSI (al momento della stesura del libro, lo standard si trovava in fase di revisione finale; si pensa che venga approvato verso la fine del 1988. Le differenze tra quanto descritto nel seguito e la versione definitiva dello standard dovrebbero essere minime). Pur avendo rilevato gli aspetti nei quali il linguaggio si è evoluto, abbiamo preferito riportare soltanto la nuova versione. Nella maggior parte dei casi, ciò non dovrebbe comportare differenze significative; il cambiamento più evidente consiste nell'introduzione della nuova forma di dichiarazione e definizione di funzione. I compilatori più recenti supportano già molte funzionalità dello standard.

Abbiamo cercato di mantenere la sinteticità della prima edizione: il C non è un linguaggio molto vasto, ed un libro voluminoso non gli si addice. Abbiamo approfondito l'esposizione di funzionalità critiche, quali i puntatori, fondamentali per la programmazione in C. Abbiamo migliorato alcuni degli esempi originali, ed in alcuni capitoli ne abbiamo aggiunti di nuovi: il trattamento delle dichiarazioni complesse viene completato con la presentazione di programmi che convertono dichiarazioni in frasi e viceversa. Inoltre, tutti gli esempi sono stati provati direttamente nella forma presentata nel testo.

L'Appendice A, il manuale di riferimento, non è lo standard, bensì il risultato del nostro tentativo di convogliare le caratteristiche essenziali dello standard in uno spazio più ristretto. Questo manuale è stato concepito per essere facilmente compreso dai programmatori, e non come definizione per i progettisti di compilatori (questo ruolo spetta allo standard stesso). L'Appendice B è un sommario delle funzionalità della libreria standard. Anch'essa dev'essere intesa come manuale di riferimento per i programmatori, non per i progettisti. L'Appendice C è un conciso elenco delle variazioni rispetto alla versione originale.

Come abbiamo detto nella prefazione alla prima edizione, il C ben si adatta ad un'esperienza in crescita e, dopo dieci anni di attività sul C, noi siamo ancora convinti che ciò sia vero. Speriamo che questo libro vi aiuti ad imparare il C e vi insegni ad usarlo bene.

Siamo profondamente grati agli amici che ci hanno aiutato a produrre questa seconda edizione. Jon Bentley, Doug McIlroy, Peter Nelson e Robe Pike hanno formulato commenti costruttivi su quasi ogni singola pagina del manoscritto. Siamo grati, per la loro attenta lettura, ad Al Aho, Dennis Allison, Joe Campbell, G. R. Emlin, Karen Fortgang, Allen Holub, Andrew Hume, Dave Kristol, John Linderman, Dave Prosser, Gene Spafford e Chris Van Wyk. Abbiamo ricevuto suggerimenti utili anche da Bill Cheswick, Mark Kernighan, Andy Koenig, Robin Lake, Tom London, Jim Reeds, Clovis Tondo e Peter Weinberger. Dave Prosser ha risposto a molte domande dettagliate sullo standard ANSI. Abbiamo usato molto il traduttore C++ di Bjarne Stroustrup per il testing locale dei nostri programmi, e Dave Kristol ci ha fornito un compilatore ANSI C per il testing finale. Rich Dreschler ci ha aiutato nella composizione. A tutti, i nostri più sinceri ringraziamenti.

Brian W. Kernighan
Dennis M. Ritchie

PREFAZIONE ALLA PRIMA EDIZIONE

Il C è un linguaggio di programmazione di uso generale, caratterizzato dalla sinteticità, da un controllo del flusso e da strutture dati avanzate, e da un vasto insieme di operatori. Il C non è un vero "linguaggio ad alto livello", e non è specializzato in alcun'area applicativa. Ma il suo essere privo di restrizioni e la sua generalità lo rendono spesso più conveniente ed efficiente di altri linguaggio supposti più potenti.

In origine, il C è stato progettato ed implementato da Dennis Ritchie su un sistema operativo UNIX, su un calcolatore DEC PDP-11. Il sistema operativo, il compilatore C ed essenzialmente tutti i programmi applicativi di UNIX (compreso il software utilizzato per preparare questo libro) sono scritti in C. Esistono anche compilatori per altre macchine, fra le quali il Sistema / 370 IBM, l'Honeywell 6000 e l'Interdata 8/32. Tuttavia, il C non è progettato per alcun hardware o sistema particolare, ed è facile scrivere programmi funzionanti, senza bisogno di alcuna modifica, su tutti i sistemi che supportano il C.

Questo libro si propone di insegnare al lettore come programmare in C. Esso contiene un'introduzione di carattere generale, alcuni capitoli relativi alle principali funzionalità ed un manuale di riferimento. La maggior parte della trattazione si basa sulla lettura, la scrittura e la revisione degli esempi, più che sulla semplice esposizione delle regole. Quasi sempre gli esempi sono costituiti non da frammenti isolati di codice, ma da programmi completi. Inoltre, tutti gli esempi sono stati provati direttamente nella forma presentata nel testo. Oltre che mostrare l'uso effettivo del linguaggio abbiamo cercato, ove possibile, di fornire algoritmi utili ed alcuni dei principi che stanno alla base di una buona metodologia di stesura del codice.

Il libro non è un manuale di introduzione alla programmazione; in esso assumiamo che il lettore possieda un certo grado di familiarità con alcuni concetti basilari, quali le variabili, le istruzioni di assegnamento, i cicli e le funzioni. Ciò nonostante, un programmatore principiante dovrebbe essere in grado di leggere il libro ed apprendere il linguaggio, magari con l'aiuto di un collega più esperto.

La nostra esperienza personale ci ha dimostrato che il C è un linguaggio piacevole, espressivo e versatile. Esso è facile da apprendere, e ben si adatta ad un'esperienza in crescita. Speriamo che questo libro vi aiuti ad imparare il C e vi insegni ad usarlo bene.

Le critiche costruttive ed i suggerimenti di molti amici hanno migliorato notevolmente questo libro, ed aumentato il nostro piacere nello scriverlo. In particolare, Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy, Bill Roome, Bob Rosin e Larry Rosler hanno letto con attenzione molte versioni dell'opera. Siamo anche riconoscenti ad Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill Plauger, Jerry Spivack, Ken Thompson e Peter Weinberger per gli utili commenti, e, infine, ringraziamo Mike Lesk e Joe Ossanna per il loro indispensabile aiuto nella composizione del libro.

Brian W. Kernighan
Dennis M. Ritchie

INTRODUZIONE

Il C è un linguaggio di programmazione di uso generale da sempre strettamente legato al sistema UNIX, sul quale è stato sviluppato, poiché sia il sistema che la maggior parte dei suoi programmi applicativi sono scritti in C. Tuttavia, questo linguaggio non è stato scritto per un particolare sistema operativo o per una particolare macchina; sebbene sia stato definito un "linguaggio di programmazione di sistema" perché adatto alla stesura di compilatori e sistemi operativi, è stato impiegato con profitto nella realizzazione di grossi programmi operanti negli ambienti più disparati.

Molte delle caratteristiche del C discendono dal linguaggio BCPL, sviluppato da Martin Richards. L'influenza del BCPL sul C passa, indirettamente, dal linguaggio B, ideato da Thompson nel 1970 per il primo sistema UNIX, sviluppato su DEC PDP-7.

I linguaggi BCPL e B sono linguaggi senza tipi. Al contrario, il C fornisce numerosi tipi di dati. I tipi fondamentali sono i caratteri ed i numeri interi e decimali. Oltre a questi, esiste un vasto insieme di tipi di dati

derivati, creati usando puntatori, vettori, strutture e union. Le espressioni sono formate da operatori ed operandi; qualsiasi espressione, compreso un assegnamento o una chiamata di funzione, può essere un'istruzione. I puntatori consentono poi un'aritmetica di indirizzamento indipendente dalla macchina.

Il C fornisce i fondamentali costrutti per il controllo del flusso, indispensabili per la stesura di programmi ben strutturati; tali costrutti sono: il raggruppamento delle istruzioni, il blocco decisionale (`if-else`), la selezione fra più alternative (`switch`), i cicli con condizione di terminazione posta in testa (`while`, `for`) ed in coda (`do`), ed infine l'uscita anticipata da un ciclo (`break`).

Le funzioni possono restituire valori appartenenti ad un tipo base oppure strutture, union o puntatori. Qualsiasi funzione può essere richiamata ricorsivamente. Le variabili locali sono "automatiche", cioè vengono ricreate ad ogni invocazione. Le definizioni di funzione non possono essere innestate, ma le variabili devono essere dichiarate secondo il metodo di strutturazione a blocchi. Le funzioni di un unico programma C possono trovarsi in file diversi che possono essere anche compilati separatamente. Le variabili possono essere dichiarate all'interno delle funzioni, al loro esterno ma visibili alle funzioni del singolo file, oppure accessibili da tutti i moduli del programma.

Una fase che precede la compilazione vera e propria, detta preprocessing, attua una sostituzione delle macro all'interno del testo del programma, esegue l'inclusione di altri file sorgente e risolve le compilazioni condizionali.

Il C è un linguaggio relativamente "a basso livello". Questa caratteristica non è peggiorativa: significa soltanto che il C tratta gli oggetti (caratteri, numeri ed indirizzi) in modo molto simile a quello utilizzato dalla maggior parte dei calcolatori; questi oggetti, infatti, possono essere combinati e spostati con l'aiuto di operatori logici ed aritmetici implementati sulle macchine reali.

Il C non fornisce operazioni per trattare direttamente oggetti compositi come le stringhe, gli insiemi, le liste od i vettori. Non ci sono operazioni che manipolano un intero vettore od una stringa, sebbene le strutture possano essere copiate come se fossero un unico oggetto. Il linguaggio non definisce alcuna funzionalità per l'allocazione di memoria, ad eccezione della definizione statica e della politica a stack utilizzata per le variabili locali alle funzioni; non esistono né uno heap né un meccanismo di garbage collection. Infine il C non prevede funzionalità esplicite di input / output; non esistono istruzioni di READ e WRITE, né metodi predefiniti di accesso ai file. Tutti questi meccanismi ad alto livello devono essere inclusi tramite esplicite chiamate di funzione. La maggior parte delle implementazioni realizzate in C ha incluso un ragionevole insieme standard di queste funzioni.

Analogamente, il C prevede soltanto un controllo del flusso molto chiaro e lineare: controlli, cicli, raggruppamenti e sottoprogrammi, ma non multiprogrammazione, operazioni parallele, sincronizzazioni o co-routine.

Sebbene l'assenza di alcune di queste funzionalità possa sembrare una grave limitazione ("Vorreste dire che, per confrontare due stringhe di caratteri, è necessario chiamare esplicitamente una funzione?"), è necessario tenere presente che mantenere il linguaggio a dimensioni ridotte comporta notevoli vantaggi. Poiché il C è relativamente piccolo, può essere descritto in uno spazio limitato e appreso velocemente. Un programmatore, quindi, può ragionevolmente attendersi di conoscere, comprendere ed usare correttamente l'intero linguaggio.

Per molti anni la definizione del C è stata quella presentata nel manuale di riferimento contenuto nella prima edizione de *Linguaggio C*. Nel 1983, l'Istituto Nazionale Americano per gli Standard (ANSI) ha costituito un comitato per la definizione aggiornata e completa del C. Il risultato di questo lavoro, lo standard ANSI, o "ANSI C", è stato approvato nel 1989. Molte delle funzionalità di questo standard sono comunque già supportate dai compilatori più recenti.

Lo standard si basa sul manuale di riferimento originale. Il linguaggio è variato soltanto in piccola parte; uno dei principali scopi dello standard, infatti, era di assicurare che i vecchi programmi continuassero a funzionare o che almeno, se ciò non fosse stato possibile, i compilatori fossero in grado di rilevare i nuovi comportamenti.

Per la maggioranza dei programmatori, la novità maggiore riguarda l'introduzione di una nuova sintassi per la dichiarazione e la definizione di funzioni. Ora una dichiarazione di funzione deve includere la descrizione degli argomenti della funzione stessa; ovviamente, anche la sintassi della definizione è stata variata di conseguenza. Quest'informazione aggiuntiva consente ai compilatori di rilevare facilmente gli errori dovuti ad

inconsistenze fra gli argomenti di chiamata e quelli della definizione; la nostra esperienza ci porta ad affermare che questa nuova funzionalità del linguaggio risulta molto utile.

Oltre a quella appena descritta, il linguaggio ha subito altre modifiche, anche se su scala minore. L'assegnamento fra strutture ed i tipi enumerativi, già disponibili in realtà, sono ora parte ufficialmente integrante del linguaggio. I calcoli in floating-point ora possono essere effettuati con precisione singola. Le proprietà della aritmetica, specialmente quelle relative ai tipi senza segno, sono state chiarite. Il preprocessore è più sofisticato. Molte di queste variazioni avranno comunque un'importanza secondaria per la maggior parte dei programmatori.

Un secondo importante contributo fornito dallo standard è la definizione di una libreria standard associata al C. Essa specifica le funzione per l'accesso al sistema operativo (per esempio per leggere e scrivere su file), per l'input e l'output formattati, per l'allocazione di memoria, per il trattamento delle stringhe ed attività affini. Una collezione di header standard fornisce un insieme uniforme di dichiarazioni di funzioni e di tipi di dati. Tutti i programmi che utilizzano questa libreria per accedere ad un sistema centrale hanno la garanzia della compatibilità di comportamento. Buona parte della libreria si basa sulla “libreria standard di I/O” del sistema UNIX, descritta nella prima edizione, che è stata utilizzata proficuamente anche su altri sistemi. Anche in questo caso, molti programmatori non noteranno il cambiamento.

Poiché i tipi di dati e le strutture di controllo fornite dal C sono direttamente supportati dalla maggioranza dei calcolatori, la libreria run-time richiesta per implementare programmi di dimensioni contenute è molto ridotta. Le funzioni della libreria standard vengono chiamate soltanto esplicitamente e quindi, se non sono necessarie, possono essere tralasciate. Quasi tutte queste funzioni sono scritte in C e, ad eccezione di quelle legate ai dettagli implementativi del sistema operativo, sono anch'esse portabili.

Anche se il C sfrutta le capacità di molti calcolatori, esso è indipendente dall'architettura della macchina. Con un minimo di attenzione è facile costruire programmi portabili, cioè programmi che possono essere eseguiti, senza bisogno di modifiche, su hardware differenti. Lo standard rende evidente questa caratteristica, e fissa un insieme di costanti che identificano la macchina sulla quale il programma è in esecuzione.

Il C non è un linguaggio fortemente tipato ma, con la sua evoluzione, anche il controllo sui tipi si è via via rafforzato. La definizione originale del C sconsigliava, ma permetteva, lo scambio tra puntatori ed interi; questa possibilità è stata eliminata, ed ora lo standard richiede dichiarazioni appropriate e conversioni esplicite, che vengono già controllate dai compilatori più recenti. Anche la nuova dichiarazione di funzione rappresenta un passo in questa direzione. I compilatori rileveranno quasi tutti gli errori sui tipi, e non sarà più consentita alcuna conversione automatica fra tipi di dati incompatibili. Tuttavia, il C ha mantenuta la filosofia di base, che assume che il programmatore sappia ciò che sta facendo; soltanto, ora si richiede che egli lo dichiari esplicitamente.

Il C, come qualsiasi altro linguaggio, ha i propri difetti. Alcuni operatori hanno la precedenza sbagliata; alcune parti della sintassi potrebbero essere migliorate. Tuttavia, esso si è dimostrato un linguaggio altamente espressivo ed efficiente, adatto ad una grande varietà di aree applicative.

Il libro è organizzato come segue. Il Capitolo 1 è un'introduzione sulla parte centrale del C. Il suo scopo è quello di consentire al lettore di iniziare a programmare nel più breve tempo possibile, perché siamo fermamente convinti che il modo migliore per imparare un linguaggio consiste nell'utilizzarlo. L'introduzione dà per scontata la conoscenza degli elemento base della programmazione; non vengono spiegati i concetti di calcolatore, compilazione, né il significato di espressioni come `n=n+1`. Anche se abbiamo cercato, ove possibile, di mostrare utili tecniche di programmazione, il libro non vuole costituire un riferimento sulle strutture dati e gli algoritmi; ogniqualvolta si è rivelato necessario fare delle scelte, abbiamo sempre preferito concentrare la nostra attenzione sul linguaggio.

I Capitoli dal 2 al 6 espongono i vari aspetti del C più dettagliatamente, ed anche più formalmente, di quanto non faccia il Capitolo 1, sebbene l'enfasi venga ancora posta sugli esempi e sui programmi completi, piuttosto che su frammenti isolati di codice. Il Capitolo 2 tratta i tipi di dati fondamentali, gli operatori e le espressioni. Il Capitolo 3 si riferisce al controllo del flusso: `if-else`, `switch`, `while`, `for`, e così via. Il Capitolo 4 riguarda le funzioni e la struttura del programma: variabili esterne, regole di scope, file sorgente multipli; sempre in questo capitolo, viene trattato anche il preprocessore. Il Capitolo 5 illustra l'aritmetica dei puntatori e degli indirizzi. Il Capitolo 6 tratta le strutture e le union.
Il Capitolo 7 descrive la libreria standard, che fornisce un'interfaccia unica verso il sistema operativo. Questa libreria è definita nello standard ANSI e dovrebbe essere presente su tutte le macchine che supportano il C,

in modo che i programmi che la usano per l'input, l'output e gli altri accessi al sistema operativo possano essere trasferiti da un sistema all'altro senza difficoltà.

Il Capitolo 8 descrive, invece, un'interfaccia tra i programmi C ed il sistema operativo UNIX, concentrandosi sull'input / output, sul file system e sull'allocazione di memoria. Anche se parte di questo capitolo si riferisce in particolare a UNIX, i programmatori che utilizzano altri sistemi dovrebbero comunque trovarvi informazioni utili quali, per esempio, una descrizione delle modalità di implementazione di una libreria standard, ed eventuali suggerimenti sulla portabilità.

L'Appendice A contiene un manuale di riferimento del linguaggio: il documento ufficiale, fornito dallo stesso ANSI, sulla sintassi e sulla semantica del C. Tale documento, comunque, si rivolge principalmente ai progettisti di compilatori. Il manuale qui riprodotto esprime la definizione del linguaggio in modo più conciso e meno rigoroso dal punto di vista formale. L'Appendice B è un sommario della libreria standard, ancora una volta rivolto agli utenti più che ai progettisti. L'Appendice C fornisce un breve riassunto delle modifiche apportate al linguaggio originale. In caso di dubbi, comunque, lo standard ed il suo compilatore sono gli elementi ai quali fare riferimento.

CAPITOLO 1

INTRODUZIONE GENERALE

Iniziamo con una breve introduzione al C. Il nostro intento è quello di illustrare gli elementi essenziali del linguaggio all'interno dei programmi reali, senza considerare, per il momento, i dettagli, le regole e le eccezioni. A questo punto dell'esposizione, la completezza e la precisione non sono i vostri principali obiettivi, che si identificano invece nella correttezza degli esercizi. Vogliamo insegnarvi, nel più breve tempo possibile, a scrivere programmi utili, e per farlo ci concentriamo sugli elementi fondamentali: variabili e costanti, aritmetica, controllo del flusso, funzioni, ed i rudimenti dell'input / output. Tralasciamo intenzionalmente, in questo capitolo, le caratteristiche del linguaggio utili nella stesura di programmi di grandi dimensioni che sono: i puntatori, le strutture, molti degli operatori del C, alcune strutture di controllo e la libreria standard.

Questo approccio ha i suoi inconvenienti. Il principale consiste nel fatto che questo libro non contiene la storia completa di tutte le caratteristiche del linguaggio e l'introduzione, essendo breve, potrebbe anche risultare poco chiara. E poiché gli esempi non usano appieno le potenzialità del C, essi non sono concisi ed eleganti come potrebbero. Abbiamo tentato di minimizzare questi difetti ma, in ogni caso, teneteli presenti. Un altro svantaggio è dato dal fatto che a volte, nei prossimi capitoli, alcuni concetti già espressi in questo verranno necessariamente ripetuti. Speriamo, in ogni caso, che queste ripetizioni risultino utili più che noiose.

In ogni caso, i programmatori già esperti dovrebbero essere in grado di estrapolare da questo capitolo gli elementi utili alle loro particolari esigenze. I principianti dovrebbero, invece, integrarlo scrivendo personalmente programmi brevi, simili a quelli degli esempi. Tutti, indistintamente, possono infine utilizzarlo come ossatura alla quale aggiungere le descrizioni più dettagliate fornite nel Capitolo 2.

1.1 Principi Fondamentali

L'unico modo per imparare un linguaggio di programmazione consiste nell'utilizzarlo. Il primo programma da scrivere è lo stesso per tutti i linguaggi:

```
Scrivi le parole
   Salve, mondo
```

Questo è il grosso ostacolo; per superarlo, dovete essere in grado di scrivere da qualche parte il testo del programma, compilarlo, caricarlo, eseguirlo e scoprire dove viene visualizzato il vostro output. Una volta superati questi dettagli tecnici, tutto, al confronto, risulterà semplice.

In C, il programma per stampare "`Salve, mondo`" è:

```
#include  <stdio.h>

main()
{
    printf("Salve, mondo\n");
}
```

Il modo per eseguire questo programma dipende dal sistema che state usando. Nel caso del sistema operativo UNIX, dovete creare il programma in un file il cui nome termini con il suffisso "`.c`", per esempio `salve.c`, e quindi compilarlo con il comando

```
cc salve.c
```

Se non avete commesso errori, quali l'omissione di un carattere o un errore di sintassi, la compilazione procederà senza messaggi e creerà un file eseguibile chiamato `a.out`. Eseguendo `a.out` con il comando

```
a.out
```

otterrete la stampa

```
Salve, mondo
```

Su altri sistemi, le regole saranno differenti: chiedete ad un esperto.

Ora, spieghiamo alcune cose relative a questo programma. Un programma C, qualunque sia la sua dimensione, è costituito da *funzioni* e da *variabili*. Una funzione contiene *istruzioni* che specificano le operazioni da effettuare, mentre le variabili memorizzano i valori usati durante l'esecuzione. Le funzioni C sono simili alle funzioni ed alle subroutine del Fortran o alle procedure e le funzioni del Pascal. Il nostro esempio è una funzione chiamata `main`. Normalmente, siete liberi di dare alle vostre funzioni i nomi che preferite, ma "`main`" è un nome speciale: il vostro programma inizia l'esecuzione a partire dalla funzione `main`. Ciò significa che ogni programma deve contenere una funzione `main`.

Di solito, `main` chiama altre funzioni che lo aiutano a svolgere il lavoro e che possono essere state scritte da voi oppure appartenere alle librerie che avete a disposizione. La prima linea del programma,

```
#include  <stdio.h>
```

dice al compilatore di includere le informazioni relative alla libreria standard di input / output; questa linea compare all'inizio di molti file sorgenti C. La libreria standard è descritta nel Capitolo 7 e nell'Appendice B.

Un metodo per comunicare dati tra funzioni consiste nel fornire alla funzione chiamata una lista di valori, detti *argomenti*, preparati dalla funzione chiamante. Le parentesi che seguono il nome della funzione racchiudono la lista di argomenti. In questo esempio, `main` è definita come una funzione che non si aspetta argomenti, come viene indicato dalla lista vuota `()`.

```
#include  <stdio.h>                  include la libreria standard

main()                               definisce una funzione main
                                     che non riceve alcun valore
                                     come argomento

{                                    le istruzioni di main sono
                                     racchiuse fra le graffe

    printf("Salve, mondo\n");        main chiama la funzione di
                                     libreria printf per stampare
                                     questa sequenza di caratteri;
                                     \n indica il new line

}
```

Il primo programma C

Le istruzioni di una funzione sono racchiuse fra parentesi graffe `{}`. La funzione `main` contiene solo una istruzione,

```
printf("Salve, mondo\n");
```

Una funzione viene chiamata con il nome, seguito da una lista di argomenti fra parentesi tonde; perciò quest'istruzione chiama la funzione `printf` con l'argomento `"Salve, mondo\n"`. `printf` è una funzione di libreria che stampa un output identificato, in questo caso, dalla stringa di caratteri racchiusa tra apici.

Una sequenza di caratteri fra doppi apici, come `"Salve, mondo\n"`, viene chiamata *stringa di caratteri* o *stringa costante*. Per il momento, l'unico uso che faremo delle stringhe di caratteri sarà come argomento di `printf` o di altre funzioni.

La sequenza `\n` presente nella stringa è la notazione usata dal C per identificare il *carattere di new line*, che quando viene stampato sposta l'output successivo sul margine sinistro della riga seguente. Tralasciando la sequenza `\n` (un esperimento che vale la pena di tentare), vedrete che dopo aver stampato l'output il cursore non cambia linea. Per includere un carattere di new line nell'argomento di `printf` dovete necessariamente usare la sequenza `\n`; se tentate di fare una cosa di questo tipo:

```
printf("Salve, mondo
");
```

il compilatore C produrrà un messaggio di errore.

`printf` non fornisce mai automaticamente un new line, quindi la costruzione di un'unica linea di output può essere effettuata tramite diverse chiamate a `printf`. Il nostro primo programma avrebbe potuto essere scritto nel modo seguente

```
#include  <stdio.h>

main()
{
    printf("Salve, ");
    printf("mondo");
    printf("\n");
}
```

ed avrebbe prodotto un identico output. Notiamo che la sequenza `\n` rappresenta un carattere singolo. Una *sequenza di escape* come `\n` fornisce un meccanismo generale ed estensibile per rappresentare caratteri difficili da vedere o invisibili. Fra le altre, il C comprende le sequenze `\t` per il carattere tab, `\b` per il backspace, `\"` per i doppi apici e `\\` per il backslash stesso. La Sezione 2.3 presenta una lista completa di queste sequenze.

Esercizio 1.1 Eseguite il programma "`Salve, mondo`" sul vostro sistema. Provate a tralasciare alcune parti del programma, per vedere quali messaggi di errore ottenete.

Esercizio 1.2 Provate a vedere cosa succede quando l'argomento di `printf` contiene una sequenza `\c`, dove `c` non è uno dei caratteri elencati sopra.

1.2 Variabili ed Espressioni Aritmetiche

Il prossimo programma usa la formula C = (5/9)(F-32) per stampare la seguente tabella delle temperature Fahrenheit e delle loro equivalenti temperature in gradi centigradi o Celsius:

```
0     -17
20    -6
40    4
60    15
80    26
100   37
120   48
140   60
160   71
180   82
200   93
220   104
240   115
260   126
280   137
300   148
```

Anche questo programma contiene la definizione di un'unica funzione chiamata `main`. Esso è più lungo del programma "`Salve, mondo`", ma non è comunque complicato. Il programma introduce alcune nuove idee, tra le quali i commenti, le dichiarazioni, le variabili, le espressioni aritmetiche, i cicli e l'output formattato.

```
#include  <stdio.h>
/* stampa la tabella Fahrenheit - Celsius
   per fahr = 0, 20, .........., 300          */
main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower=0;            /* limite inferiore della tabella */
    upper=300;          /* limite superiore */
    step=20;            /* incremento */
```

```
    fahr=lower;
    while (fahr<=upper)
    {
        celsius=5*(fahr-32)/9;
        printf("%d\t%d\n, fahr, celsius);
        fahr=fahr+step;
    }
}
```

Le due linee

```
/* stampa la tabella Fahrenheit - Celsius
   per fahr = 0, 20, ........., 300          */
```

sono dei *commenti* che, in questo caso, spiegano brevemente ciò che il programma fa. Tutti i caratteri compresi fra `/*` e `*/` vengono ignorati dal compilatore; questi commenti possono essere usati liberamente per rendere più leggibile un programma. Essi possono apparire in tutte le posizioni nelle quali possono comparire spazi bianchi, caratteri di tabulazione o new line.

In C, prima di poter essere utilizzate, tutte le variabili devono essere dichiarate, normalmente all'inizio della funzione, prima di una qualsiasi istruzione eseguibile. Una *dichiarazione* denuncia le proprietà delle variabili; essa consiste in un nome di tipo ed in una lista di variabili, come

```
int fahr, celsius;
int lower, upper, step;
```

Il tipo `int` indica che le variabili elencate di seguito sono degli interi, mentre `float` indica variabili decimali, cioè che possono avere una parte frazionaria. Il range dei valori che le variabili di tipo `int` e `float` possono assumere dipende dalla macchina che state usando; potete trovare interi a 16-bit, compresi fra -32768 e +32767, oppure interi a 32-bit. Un numero `float` è, di solito, una quantità a 32 bit, con almeno 6 bit significativi e può variare tra 10^(-38) e 10^(+38).

Oltre ai tipi `int` e `float`, il C prevede altri tipi di dati fondamentali, tra i quali:

`char`	carattere: un singolo byte
`short`	intero corto
`long`	intero lungo
`double`	numero decimale in doppia precisione

Anche le dimensioni di questi oggetti dipendono dalla macchina. Esistono poi i *vettori*, le *strutture* e le *union* di questi tipi fondamentali, i *puntatori* ad essi, e le *funzioni* che li restituiscono: tutti elementi che illustreremo al momento opportuno.

L'esecuzione del programma di conversione delle temperature inizia con alcune *istruzioni di assegnamento*

```
lower = 0;
upper = 300;
step = 20;
fahr = lower;
```

che assegnano alle variabili i loro valori iniziali. Le singole istruzioni sono terminate da un punto e virgola.

Ogni linea della tabella viene calcolata nello stesso modo, perciò usiamo un ciclo che si ripete per ogni riga di output; questo è dunque lo scopo del ciclo di `while`

```
while (fahr<=upper)
{
    ............
}
```

Il `while` opera nel seguente modo: viene valutata la condizione tra parentesi. Se è vera (cioè, in questo caso, se `fahr` è minore o uguale ad `upper`), viene eseguito il corpo del ciclo (cioè le istruzioni racchiuse tra

parentesi graffe). Quindi la condizione viene ricontrollata e, se è vera, il corpo del ciclo viene rieseguito. Quando la condizione diventa falsa (fahr risulta maggiore di upper) il ciclo termina e l'esecuzione riprende dalla prima istruzione successiva al ciclo. Nel nostro caso, non esistendo altre istruzioni, il programma termina.

Il corpo di un while può essere costituito da una o più istruzioni fra parentesi graffe, come nel caso del programma di conversione delle temperature, oppure da una singola istruzione senza parentesi, come mostra lo esempio seguente

```
while (i<j)
    i=2*j;
```

In entrambi i casi, in questo libro indenteremo di un carattere di tabulazione le istruzioni controllate dal while, in modo che risulti evidente quali istruzioni compongono il corpo del ciclo. L'indentazione enfatizza la struttura logica di un programma. Anche se i compilatori C ne ignorano l'aspetto esteriore, i programmi con un'indentazione ed una spaziatura corretta risultano più leggibili.

Vi esortiamo dunque a scrivere una sola istruzione per riga, e ad usare gli spazi bianchi fra gli operatori, in modo da evidenziarne il raggruppamento. La posizione delle parentesi è meno importante, sebbene la gente vi ponga molta attenzione. In proposito, noi abbiamo scelto una delle numerose possibilità. Scegliete lo stile che preferite, purché lo usiate in modo consistente.

La maggior parte dell'elaborazione viene svolta all'interno del ciclo. La temperatura Celsius viene calcolata ed assegnata alla variabile celsius con l'istruzione

```
celsius=5*(fahr-32)/9
```

La ragione per cui abbiamo scelto di moltiplicare per 5 e poi dividere per 9, invece di moltiplicare direttamente per 5/9, è che in C, come in molti altri linguaggi, la divisione fra interi effettua un *troncamento*: qualsiasi parte frazionaria viene scartata. Poiché 5 e 9 sono due numeri interi, 5/9 verrebbe troncato a zero e, di conseguenza, tutte le temperature Celsius risulterebbero nulle.

Questo esempio illustra anche in maggiore dettaglio il modo in cui lavora la funzione printf. printf è una funzione di uso generale per la stampa di output formattato, che descriveremo per esteso nel Capitolo 7. Il suo primo argomento è una stringa di caratteri da stampare, nella quale ogni % indica il punto in cui devono essere sostituiti, nell'ordine, il secondo, il terzo e via via tutti gli argomenti seguenti; i caratteri immediatamente successivi ad un % indicano la forma nella quale l'argomento dev'essere stampato. Per esempio, %d specifica un argomento intero, quindi l'istruzione

```
printf("%d\t%d\n", fahr, celsius);
```

provoca la stampa dei due valori interi contenuti nelle variabili fahr e celsius, separati da un carattere di tabulazione (\t).

Ogni occorrenza del carattere % nel primo argomento è associata al corrispondente argomento di printf, a partire dal secondo: perché non si verifichino errori, la corrispondenza deve riguardare sia il numero che il tipo degli argomenti.

Per inciso, notiamo che printf non appartiene al linguaggio C; il C, di per se stesso, non prevede alcuna definizione di funzioni di input / output. printf è soltanto un'utile funzione presente nella libreria standard delle funzioni normalmente accessibili da programmi C. Il comportamento di printf è definito nello standard ANSI e, di conseguenza, le sue proprietà dovrebbero essere le stesse con tutti i compilatori e le libre-rie conformi allo standard.

Per poterci concentrare sul C, rimandiamo al Capitolo 7 ulteriori dissertazioni riguardanti l'input / output. In particolare, rinviamo fino ad allora la trattazione dell'input formattato. Se dovete ricevere dei numeri in input, leggete la discussione sulla funzione scanf, nella Sezione 7.4. scanf è simile a printf, ma si occupa di leggere l'input, piuttosto che di scrivere l'output.

Nel programma di conversione delle temperature esistono due problemi. Il più semplice è dato dal fatto che la formattazione dell'output potrebbe essere migliorata, allineando a destra i numeri stampati. La soluzione è

semplice; aggiungendo ad ogni sequenza `%d` della `printf` un'ampiezza, i numeri in output verranno incolonnati a destra. Per esempio, potremmo scrivere

```
printf("%3d %6d\n", fahr, celsius);
```

per stampare il primo numero di ogni linea in un campo di tre cifre, ed il secondo in un campo di sei, ottenendo il seguente risultato:

```
   0      -17
  20       -6
  40        4
  60       15
  80       26
 100       37
....       ..
```

Il problema principale, invece, è dato dal fatto che abbiamo usato l'aritmetica fra interi e, di conseguenza, le temperature Celsius risultano poco accurate; per esempio, 0F corrispondono, in realtà, a -17.8 C, e non a -17 C.

Per ottenere misure più precise, dovremmo utilizzare l'aritmetica in floating-point, e per farlo dobbiamo apportare alcune modifiche al programma. Eccone dunque una seconda versione:

```
#include  <stdio.h>

/* stampa una tabella Fahrenheit - Celsius per
   fahr = 0, 20, ........, 300; versione con floating-point */

main()
{
    float fahr, celsius;
    int lower, upper, step;

    lower=0;            /* limite inferiore della tabella */
    upper=300;          /* limite superiore */
    step=20;            /* incremento */

    fahr=lower;
    while (fahr<=upper)
    {
        celsius=(5.0/9.0)*(fahr-32);
        printf("%3.0f  %6.1f\n", fahr, celsius);
        fahr=fahr+step;
    }
}
```

Questo programma è molto simile a quello precedente, ma le variabili `fahr` e `celsius` sono state dichiarate `float`, e la formula di conversione è scritta in modo più intuitivo. Nella versione precedente, non potevamo usare l'espressione `5/9` perché la divisione intera avrebbe troncato a zero il risultato. Tuttavia, un punto decimale in una costante indica che essa è un numero frazionario, quindi `5.0/9.0` non viene troncato, perché è un quoziente tra due numeri decimali.

Se un operatore aritmetico ha degli operandi interi, l'operazione eseguita è intera. Tuttavia, se un operatore ha un argomento intero ed uno decimale, prima di eseguire l'operazione l'intero viene convertito in un decimale. Se avessimo scritto `fahr-32`, il valore `32` sarebbe stato trasformato automaticamente in un floating-point. Ciò nondimeno, scrivere delle costanti con dei punti decimali espliciti anche quando esse hanno valori interi ne enfatizza, agli occhi del lettore, la natura frazionaria.

Le regole dettagliate per le conversioni degli interi in numeri frazionari sono date nel Capitolo 2. Per ora, notiamo soltanto che l'assegnamento

```
fahr=lower;
```

ed il test

```
while (fahr<=upper)
```

lavorano operando la conversione degli interi in numeri decimali.

La specifica di conversione `%3.0f` nella `printf` dice che un numero frazionario (`fahr`) dev'essere stampato in un campo di almeno tre caratteri, senza punto decimale e senza cifre frazionarie. `%6.1f` dice invece che un altro numero (`celsius`) dev'essere stampato in un campo di almeno sei caratteri, con una cifra dopo il punto decimale. L'output ottenuto sarà del tipo:

```
  0    -17.8
 20     -6.7
 40      4.4
...      ...
```

L'ampiezza e la precisione, in una specifica di conversione, possono essere tralasciate: `%6f` significa che il numero dev'essere in un campo di almeno sei caratteri; `%.2f` indica la presenza di due cifre dopo il punto decimale, ma non pone vincoli sull'ampiezza del campo; `%f`, infine, dice semplicemente che il numero deve essere stampato in floating-point.

`%d`	stampa un intero decimale
`%6d`	stampa un intero decimale in un campo di almeno sei caratteri
`%f`	stampa un numero frazionario
`%6f`	stampa un numero frazionario in un campo di almeno sei caratteri
`%.2f`	stampa un numero frazionario, con due cifre dopo il punto decimale
`%6.2f`	stampa un numero frazionario, in un campo di almeno sei caratteri e con almeno due cifre dopo il punto decimale

Inoltre, `printf` riconosce anche le sequenze `%o` per le stampe in ottale, `%x` per quelle in esadecimale, `%s` per le stringhe di caratteri e `%%` per il carattere `%` stesso.

Esercizio 1.3 Modificate il programma di conversione delle temperature in modo che stampi un'intestazione.

Esercizio 1.4 Scrivete un programma che stampi una tabella di corrispondenza fra temperature Celsius e Fahrenheit.

1.3 L'Istruzione FOR

Esistono molti modi di scrivere un programma che assolva un particolare compito. Tentiamo dunque di scrivere una variante del programma di conversione delle temperature.

```
#include  <stdio.h>

/* stampa la tabella Fahrenheit - Celsius */
main()
{
    int fahr;

    for (fahr=0; fahr<=300; fahr=fahr+20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Questo programma produce gli stessi risultati, pur apparendo molto diverso dal precedente. Una delle variazioni principali è l'eliminazione di molte variabili; è rimasta soltanto la variabile `fahr`, che abbiamo dichiarato come `int`. I limiti inferiore e superiore (`lover` ed `upper`) e la granularità (`step`) appaiono soltanto come costanti nell'istruzione `for`, che è anch'essa un costrutto nuovo; infine, l'espressione per il calcolo dei gradi Celsius compare come terzo argomento della `printf`, invece che come istruzione separata di assegnamento.

Quest'ultimo cambiamento è un esempio di applicazione di una regola generale: in ogni contesto in cui è consentito l'uso del valore di una variabile di un certo tipo, è possibile utilizzare anche espressioni di quel tipo

ma più complesse. Poiché il terzo argomento della `printf` dev'essere un valore frazionario, per consistenza con la specifica di conversione `%6.1f`, esso può essere una qualsiasi espressione di tipo frazionario.

L'istruzione `for` è un ciclo, una generalizzazione del `while`. Confrontandola con il `while` della prima versione del programma, le operazioni che essa effettua dovrebbero risultare chiare. All'interno delle parentesi vi sono tre parti, separate da un punto e virgola. La prima parte, l'inizializzazione

```
fahr=0
```

viene eseguita una sola volta, prima di entrare nel ciclo. La seconda parte è la condizione di controllo del ciclo:

```
fahr<=300
```

Questa condizione viene valutata; se è vera, viene eseguito il corpo del ciclo (che nel nostro caso è costituito soltanto da una `printf`). Quindi viene eseguito l'incremento

```
fahr=fahr+20
```

e la condizione viene nuovamente testata. Il ciclo termina quando la condizione diventa falsa. Come nel caso del `while`, anche il corpo del `for` può essere composto da una singola istruzione o da un gruppo di istruzioni racchiuse tra parentesi graffe. L'inizializzazione, la condizione e l'incremento possono essere espressioni di qualsiasi tipo.

La scelta tra un `while` ed un `for` è arbitraria, e cade, in genere su quello fra i due risulta di volta in volta più chiaro. Normalmente, il `for` è più adatto a cicli nei quali l'inizializzazione e l'incremento sono delle istruzioni singole e logicamente correlate, poiché è più compatto del `while` e raggruppa in un'unica posizione tutte le istruzioni di controllo del ciclo.

Esercizio 1.5 Modificate il programma di conversione delle temperature in modo che stampi la tabella in ordine inverso, cioè partendo dai 300 gradi e scendendo fino a 0.

1.4 Costanti Simboliche

Un'osservazione finale prima di abbandonare definitivamente il programma di conversione delle temperature. Mantenere, all'interno del codice, numeri espliciti non è una buona abitudine; essi, infatti, non forniscono alcuna informazione ad un eventuale lettore, e risultano difficili da modificare in modo sistematico. Un mezzo per rendere più maneggevoli questi numeri consiste nell'associarli a dei nomi significativi. Una linea che inizia con `#define` definisce un *nome simbolico*, o *costante simbolica*, come una particolare stringa di caratteri:

```
#define  nome           testo da sostituire
```

Dopo una linea di questo tipo, tutte le occorrenze di *nome* (purché non siano racchiuse fra apici e non facciano parte di un'altra stringa) vengono rimpiazzate con il corrispondente *testo da sostituire*. *nome* ha la stessa forma di un nome di variabile: una sequenza di lettere e cifre inizia con una lettera. Il *testo da sostituire* può essere una qualsiasi sequenza di caratteri; esso, cioè, non deve necessariamente essere un numero.

```
#include <stdio.h>

#define  LOWER      0   /* limite inferiore della tabella */
#define  UPPER    300   /* limite superiore */
#define   STEP     20   /* incremento */

/* stampa la tabella Fahrenheit - Celsius */
main()
{
    int fahr;

    for (fahr=LOWER; fahr<=UPPER; fahr=fahr+STEP)
        printf("%3d  %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Le quantità LOWER, UPPER e STEP sono costanti simboliche, non variabili, che in quanto tali non appaiono nelle dichiarazioni. Per meglio distinguerli dai nomi delle variabili, normalmente scritti in lettere minuscole, i nomi delle costanti simboliche vengono convenzionalmente scritti a caratteri maiuscoli, così da favorire la leggibilità del programma. Notiamo che, in fondo ad una linea del tipo #define, non viene posto il punto e virgola.

1.5 Input / Output di Caratteri

Consideriamo ora una famiglia di programmi correlati, tutti relativi al trattamento dei caratteri. Troverete ben presto che alcuni di essi non sono altro che le versioni espanse dei prototipi sin qui discussi.

Il modello di input / output supportato dalla libreria standard è molto semplice. L'input o l'output di un testo, qualsiasi siano la sua sorgente e la destinazione, viene considerato come un flusso di caratteri. Un *flusso di testo* è una sequenza di caratteri divisi in linee; ogni linea consiste in zero o più caratteri seguiti da un new line. La conformità di ogni flusso di input e di output a questo modello è responsabilità della libreria; utilizzandola, il programmatore C non ha bisogno di preoccuparsi di come le linee vengono rappresentate all'esterno del programma.

Tra le diverse funzioni per la lettura e la scrittura di un carattere, fornite dalla libreria standard, le più semplici sono getchar e putchar. Ogni volta che viene chiamata, getchar legge il *prossimo carattere di input* da un flusso di caratteri e lo restituisce come suo valore. Cioè, dopo

```
c=getchar();
```

la variabile c contiene il prossimo carattere di input. Normalmente, i caratteri letti vengono inseriti dalla tastiera; l'input da file verrà illustrato nel Capitolo 7.

La funzione putchar stampa un carattere ogni volta che viene invocata:

```
putchar(c)
```

stampa il contenuto della variabile intera c come carattere, normalmente sul video del terminale. Le chiamate a putchar e printf possono venire intercalate: l'output apparirà nell'ordine in cui le chiamate stesse vengono effettuate.

1.5.1 Copia tra i File

Conoscendo getchar e putchar, è possibile scrivere una quantità sorprendente di programmi utili, senza alcuna nozione aggiuntiva sull'input / output. L'esempio più semplice è quello di un programma che copia il suo input sul suo output, procedendo al ritmo di un carattere per volta:

```
leggi un carattere
while (il carattere è diverso da new line)
    stampa il carattere appena letto
    leggi un carattere
```

La conversione in C è la seguente

```
#include  <stdio.h>

/* copia l'input sull'output; prima versione */
main()
{
    int c;

    c=getchar();
    while (c!=EOF)
    {
        putchar(c);
        c=getchar();
    }
```

```
}
```

L'operatore relazione `!=` significa “diverso da”.

Ciò che sulla tastiera e sul video appare come carattere viene memorizzato, come ogni altro oggetto, sotto forma di una stringa di bit (bit-pattern). Il tipo `char` specifica proprio questo tipo di caratteri ma, al suo posto, è possibile utilizzare uno qualsiasi dei tipi interi. Noi abbiamo usato `int` per una sottile ma importante ragione.

Il problema consiste nel distinguere la fine dell'input dei dati validi. La soluzione è data dal fatto che `get-char`, quando non c'è più input, ritorna un valore particolare, che non può venire confuso con alcun altro carattere reale. Questo valore è chiamato `EOF`, che significa “End Of File”, cioè “Fine Del Testo”. Noi dobbiamo dichiarare la variabile `c` di un tipo in grado di contenere uno qualsiasi dei valori che `getchar` può ritornare. Non possiamo utilizzare il tipo `char` perché `c` dev'essere sufficientemente grande per conte-nere, oltre ad ogni carattere possibile, anche l'`EOF`. Per questo motivo, usiamo il tipo `int`.

`EOF` è un intero definito in `<stdio.h>`, ma il suo valore specifico non è significativo, purché sia diverso dal valore di qualsiasi altro `char`. Usando una costante simbolica, ci assicuriamo che nessuna parte del programma dipenda da questo valore specifico.

Il programma di copia potrebbe essere scritto in modo più conciso da un programmatore esperto. In C, ogni assegnamento, come

```
c=getchar();
```

costituisce un'espressione con un particolare valore, che è quello della parte sinistra dopo l'assegnamento. Questo significa che ogni assegnamento può fare parte di un'espressione più complessa. Se l'assegnamento di un carattere alla variabile `c` viene inserito nel test di un ciclo di `while`, il programma di copia può essere scritto nella seguente forma:

```
#include  <stdio.h>

/* copia l'input sull'output; seconda versione */
main()
{
    int c;
    while ((c=getchar()!=EOF)
        putchar(c);
}
```

Il `while` preleva un carattere, lo assegna a `c`, quindi controlla che il carattere fosse diverso dal segnale di End Of File. Se ciò si verifica, viene eseguito il corpo del `while`, che stampa il carattere in questione. Quindi, il ciclo si ripete. Quando viene raggiunta la fine dell'input, il `while`, e di conseguenza il `main`, terminano.

Questa versione accentra l'input (ora c'è soltanto un punto di chiamata a `getchar`) ed abbrevia il programma. Il codice che ne deriva è più compatto e, una volta appreso il principio su cui si basa, anche più leggibile. Vedrete spesso questo tipo di scrittura (è comunque possibile che ci si lasci fuorviare e si scriva codice incomprensibile; una tendenza, questa, che cercheremo di dominare).

Le parentesi intorno all'assegnamento, all'interno della condizione di riciclo, sono necessarie. La *precedenza* dell'operatore `!=` è infatti superiore a quella dell'operatore `=`, il che significa che, in assenza di parentesi, il test relazionale `!=` verrebbe eseguito prima dell'assegnamento `=`. In altre parole, l'istruzione

```
c=getchar()!=EOF
```

equivale a

```
c=(getchar()!=EOF)
```

Quest'istruzione ha l'effetto, indesiderato, di assegnare a `c` il valore 0 oppure 1, in base al fatto che `get-char` abbia incontrato o meno la fine del file (chiarimenti in proposito verranno presentati nel Capitolo 2).

Esercizio 1.6 Verificate che l'espressione `getchar() != EOF` è sempre uguale a 0 oppure ad 1.

Esercizio 1.7 Scrivete un programma che stampi il valore di `EOF`.

1.5.2 Conteggio dei Caratteri

Il prossimo programma conta i caratteri, ed è simile al programma di copia.

```
#include  <stdio.h>

/* conta i caratteri; prima versione */
main()
{
    long nc;

    nc=0;
    while (getchar()!=EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

L'istruzione

```
++nc;
```

presenta un nuovo operatore, `++`, che significa *incremento di uno*. Lo stesso risultato si otterrebbe scrivendo `nc=nc+1`, ma `++nc` è più conciso e, spesso, più efficiente. Esiste anche un corrispettivo per il decremento di uno: l'operatore `--`. Gli operatori `++` e `–` possono essere sia prefissi (`++nc`) che postfissi (`nc++`); come vedremo nel Capitolo 2, queste due forme assumono valori diversi all'interno di un'espressione, anche se sia `++nc` che `nc++` incrementano di uno la variabile `nc`. Per il momento, ci limiteremo ad utilizzare la forma prefissa.

Il programma di conteggio dei caratteri accumula il totale parziale di una variabile di tipo `long`, invece che in un `int`. Gli interi `long` sono costituiti da almeno 32 bit. Anche se su molte macchine gli `int` ed i `long` hanno la stessa ampiezza, su altre gli `int` occupano 16 bit, e possono quindi assumere un valore massimo di 32767; un contatore intero andrebbe quindi in overflow troppo facilmente. La specifica di conversione `%ld` dice alla `printf` che l'argomento corrispondente è un intero di tipo `long`.

Utilizzando un `double` (`float` in doppia precisione) è possibile conteggiare anche numeri maggiori. Nella seconda versione di questo programma, useremo anche un `for` al posto del `while`, per illustrare un altro modo di scrivere il ciclo.

```
#include  <stdio.h>

/* conta i caratteri; seconda versione */
main()
{
    double nc;

    for (nc=0; getchar()!=EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

`printf` usa `%f` sia per i `double` che per i `float`; `%.0f` elimina la stampa del punto decimale e della parte frazionaria, che è nulla.

Il corpo di questo ciclo di `for` è vuoto, perché tutta l'elaborazione viene effettuata nelle parti di test e di incremento. Tuttavia, le regole grammaticali del C richiedono che un'istruzione di `for` possieda sempre un corpo. Il punto e virgola isolato, chiamato *istruzione nulla*, soddisfa questo requisito. L'abbiamo posto su una linea separata per renderlo visibile.

Prima di abbandonare il programma del conteggio dei caratteri, osserviamo che, se l'input non contiene caratteri, la condizione del `while` (o del `for`) fallisce alla prima chiamata di `getchar`, ed il programma produce uno zero, che è il risultato corretto. Questo è molto importante. Uno dei vantaggi del `while` e del `for` è quello di controllare la condizione prima di eseguire il corpo del ciclo stesso. Se non ci sono azioni da intraprendere, nulla viene eseguito, anche se ciò significa non entrare mai nel corpo del ciclo. I programmi dovrebbero sempre gestire il caso di un input nullo. Le istruzioni `while` e `for` aiutano a fare sì che il programma si comporti in modo ragionevole anche nei casi limite.

1.5.3 Conteggio delle Linee

Il prossimo programma conta le linee. Come abbiamo già detto, la libreria standard assicura che un flusso di input appaia come una sequenza di linee, ognuna delle quali è terminata dal carattere di new line. Perciò, contare le linee equivale a contare i new line:

```
#include  <stdio.h>

/* conta le linee in input */
main()
{
    int c, nl;

    nl=0;
    while ((c=getchar() != EOF)
        if (c==‘\n’)
            ++nl;
    printf(“%d\n”, nl);
}
```

Questa volta il corpo del `while` è costituito da un `if`, che a sua volta controlla l'incremento `++nl`. L'istruzione `if` controlla la condizione fra parentesi e, se questa è vera, esegue l'istruzione (o il gruppo fra parentesi graffe) immediatamente successiva. Nell'esempio, abbiamo utilizzato un altro livello di indentazione, per evidenziare il controllo delle istruzioni le une sulle altre.

L'operatore `==` è la notazione C per "uguale a" (è il corrispettivo degli operatori `=` del Pascal e `.EQ.` del Fortran). Questo simbolo viene usato per distinguere il test di uguaglianza dall'assegnamento, che in C è indicato dall'operatore `=`. Un'osservazione: i programmatori inesperti scrivono spesso `=` al posto di `==`. Come vedremo nel Capitolo 2, il risultato di questo errore è, di solito, un'espressione legale che, tuttavia, produce risultati diversi da quelli attesi.

Un carattere scritto tra singoli apici rappresenta un valore intero, uguale al valore numerico del carattere allo interno del set di caratteri della macchina. Tale carattere viene detto *costante di tipo numerico*, anche se non è altro che un modo diverso di scrivere i piccoli interi. Così, per esempio, `‘A’` è una costante di tipo carattere; nel codice ASCII il suo valore numerico è 65, la rappresentazione interna del carattere `A`. naturalmente, piuttosto che 65, è preferibile utilizzare la notazione `‘A’`: il suo significato è ovvio ed indipendente dal set di caratteri utilizzato dalla macchina.

Le sequenze di escape usate nelle stringhe costanti sono legati anche nelle costanti di tipo carattere, quindi `‘\n’` indica il valore del carattere di new line, che in ASCII è 10. Notate che `‘\n’` è un carattere singolo e, nelle espressioni, equivale ad un intero; al contrario, `“\n”` è una stringa costante che, casualmente, contiene un solo carattere. Le peculiarità delle stringhe rispetto ai caratteri vengono illustrate nel Capitolo 2.

Esercizio 1.8 Scrivete un programma che conti gli spazi, i caratteri di tabulazione ed i new line.

Esercizio 1.9 Scrivete un programma che copi il suo input sul suo output, sostituendo una stringa di uno o più spazi con uno spazio singolo.

Esercizio 1.10 Scrivete un programma che copi il suo input sul suo output, sostituendo ogni carattere di tabulazione con `\t`, ogni backspace con `\b` ed ogni backslash con `\\`. Questo visualizza in modo univoco i caratteri di tabulazione ed i backspace.

1.5.4 Conteggio delle Parole

Il quarto della nostra serie di utili programmi conta le linee, le parole ed i caratteri, assumendo che una parola sia una qualsiasi sequenza di caratteri priva di spazi, di caratteri di tabulazione e di new line. Quella che segue è una versione semplificata del programma UNIX `wc`.

```
#include  <stdio.h>

#define IN   1           /* all'interno di una parola */
#define OUT  0           /* all'esterno di una parola */

/* conta linee, parole e caratteri in input */
main()
{
    int c, nl, nw, nc, state;

    state=OUT;
    nl=nw=nc=0;
    while ((c=getchar()) != EOF)
    {
         ++nc;
         if (c=='\n')
             ++nl;
         if (c==' '  ||  c=='\n'  ||  c=='\t')
             state=OUT;
         else if (state==OUT)
         {
             state=IN;
             ++nw;
         }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

Ogni volta che il programma incontra il primo carattere di una parola, incrementa di uno il contatore delle parole. La variabile `state` registra il fatto che il programma, in un certo istante, si trovi o meno all'interno di una parola; inizialmente essa vale `OUT`, che significa "non siamo all'interno di una parola". Ai valori numerici 1 e 0 preferiamo le costanti simboliche `IN` ed `OUT`, perché rendono il programma più leggibile. In un programma scarno come questo, un simile accorgimento può sembrare superfluo ma, in programmi più complessi, lo aumento di chiarezza che comporta supera di gran lunga lo sforzo derivante dall'utilizzare sin dall'inizio questa notazione. Scoprirete, inoltre, che è molto più agevole apportare modifiche estese a programmi nei quali i numeri espliciti compaiono soltanto come costanti simboliche.

La linea

```
nl=nw=nc=0;
```

pone a zero tre variabili. Questa non è un'eccezione, bensì una conseguenza del fatto che un assegnamento è un'espressione con un suo valore, e che gli assegnamenti sono associativi da destra a sinistra. È come se avessimo scritto

```
nl=(nw=(nc=0));
```

Il simbolo `||` identifica l'operatore logico OR, perciò la linea

```
if (c==' '  ||  c=='\n'  ||  c=='\t')
```

va intesa come "se `c` è uno spazio, *oppure* un new line, *oppure* un carattere di tabulazione" (ricordiamo che la sequenza di escape `\t` è la rappresentazione visibile del carattere di tabulazione). Esiste un operatore corrispondente, `&&`, per l'operazione di AND; la sua precedenza è immediatamente superiore a quella di `||`. Le espressioni connesse da `&&` e `||` vengono valutate da sinistra a destra, e la valutazione si blocca non appena viene determinata la verità o la falsità dell'intera espressione. Se `c` è uno spazio, non è necessario controllare che esso sia un carattere di tabulazione o un new line, quindi questi controlli non vengono effettuati. Quest'osservazione, che in questo caso non è particolarmente importante, diventa determinante in situazioni più complesse, come vedremo in seguito.

L’esempio mostra anche un `else`, che specifica un’azione alternativa da intraprendere se la condizione associata ad un `if` risulta falsa. La forma generale è:

```
if (espressione)
    istruzione_1
else
    istruzione_2
```

Una ed una sola delle due istruzioni associate ad un `if-else` viene eseguita. Se *espressione* è vera, viene eseguita *istruzione_1*; in caso contrario, viene eseguita *istruzione_2*. Ogni istruzione può essere un’istruzione singola o un gruppo, racchiuso in parentesi graffe. Nel programma di conteggio delle parole, l’istruzione dopo l’`else` è un `if` che controlla due istruzioni fra parentesi graffe.

Esercizio 1.11 Come potreste controllare la correttezza del programma di conteggio delle parole? Quali sequenze di input sarebbero più adatte a rilevare eventuali errori?

Esercizio 1.12 Scrivete un programma che stampi il suo input con una parola per linea.

1.6 Vettori

Scriviamo ora un programma che conta le occorrenze delle diverse cifre, dei caratteri di spaziatura (spazio, carattere di tabulazione e new line), e di tutti gli altri caratteri. Questa suddivisione è artificiosa, ma ci consente di evidenziare alcune caratteristiche di un programma C.

Poiché i possibili tipi di input sono dodici, per memorizzare le occorrenze delle dieci cifre useremo un vettore (array), anziché dieci variabili distinte. Ecco dunque una versione del programma:

```
#include  <stdio.h>

/* conta le cifre, le spaziature e gli altri caratteri */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite=nother=0;
    for (i=0; i<10; ++i)
         ndigit[i]=0;

    while ((c=getchar()) != EOF)
         if (c>=‘0’  &&  c<=‘9’)
             ++ndigit[c-‘0’];
         else if (c==‘ ‘  ||  c==‘\n’  || c==‘\t’)
             ++nwhite;
         else
             ++nother;

    printf(“cifre =”);
    for (i=0; i<10; ++i)
         printf(“  %d”, ndigit[i]);
    printf(“, spaziature = %d, altri = %d\n, nwhite, nother);
}
```

L’output di questo programma, eseguito con il suo stesso testo in input, è

```
Cifre = 9 3 0 0 0 0 0 0 0 1, spaziature = 123, altri = 345
```

La dichiarazione

```
int ndigit[10];
```

definisce `ndigit` come un vettore di dieci interi. Poiché, in C, gli indici di un vettore partono sempre da zero, gli elementi di `ndigit` sono `ndigit[0]`, `ndigit[1]`, `. . . .`, `ndigit[9]`. Questa proprietà si riflette nella struttura dei cicli `for` che inizializzano e stampano il vettore.

Un indice può essere una qualsiasi espressione intera e, in quanto tale, può contenere costanti e variabili intere come, nel nostro caso, la variabile `i`.

Questo particolare programma si basa sulle proprietà di rappresentazione alfanumerica delle cifre. Per esempio, il test

```
if (c>='0'  &&  c<='9') ......
```

stabilisce se il carattere `c` è o meno una cifra. Se lo è, il valore numerico di quella cifra è

```
c='0'
```

Questa vale soltanto se `'0'`, `'1'`, `. . . .`, `'9'` possiedono valori crescenti e consecutivi. Fortunatamente, ciò è vero per tutti i set di caratteri.

Per definizione, i `char` non sono altro che dei piccoli interi; questo comporta che, nelle espressioni aritmetiche, essi siano identici agli `int`. Una simile convenzione è molto vantaggiosa; per esempio, `c-'0'` è una espressione intera, con un valore compreso fra 0 e 9 corrispondente al carattere da `'0'` a `'9'` memorizzato in `c`: tale valore è, dunque, utilizzabile anche come indice del vettore `ndigit`.

L'identificazione della classe di appartenenza del carattere (spaziatura, cifra o altro), viene effettuata con la sequenza

```
if (c>='0'  &&  c<='9')
    ++ndigit[c-'0'];
else if (c==' '  ||  c=='\n'  ||  c=='\t')
    ++nwhite;
else
    ++nother;
```

La struttura

```
if (condizione_1)
    istruzione_1
else if (condizione_2)
    istruzione_2
.....
    .......
else
    istruzione_n
```

viene utilizzata spesso come mezzo per effettuare una scelta fra più possibilità. A partire dalla prima in alto, le diverse *condizioni* vengono valutate sequenzialmente, fino a che non se ne trovi una vera; a questo punto, viene eseguita l'istruzione corrispondente alla condizione vera, dopodiché si esce dall'intera struttura (ogni *istruzione* può essere costituita da più istruzioni racchiuse fra parentesi graffe). Se nessuna delle condizioni previste è soddisfatta, l'istruzione eseguita è quella controllata dall'ultimo `else`, se esiste. Se l'`else` finale e la corrispondente istruzione vengono omessi, come nel programma per il conteggio delle parole, non viene eseguita alcuna azione particolare. Il numero dei gruppi

```
else if (condizione)
    istruzione
```

compresi tra il primo `if` è l'ultimo `else` è arbitrario.

Per motivi di chiarezza, è conveniente rappresentare il costrutto `else-if` nel modo illustrato dal programma; se ogni `if` fosse indentato rispetto all'`else` precedente, una lunga sequenza di alternative sconfinerebbe oltre il margine destro della pagina.

L'istruzione `switch`, che verrà discussa nel Capitolo 3, fornisce un modo alternativo di realizzare un simile costrutto. L'uso di quest'istruzione è particolarmente indicato quando la condizione consiste nel determinare se una particolare espressione possiede valori all'interno di un certo insieme di costanti. A scopo di confronto, nella Sezione 3.4 presenteremo una versione di questo programma nella quale viene usata l'istruzione `switch`.

Esercizio 1.13 Scrivete un programma che stampi un istogramma della lunghezza delle parole che riceve in input. Un istogramma a barre orizzontali è più facile da ottenere di uno a barre verticali.

Esercizio 1.14 Scrivete un programma che stampi un istogramma delle frequenze dei diversi caratteri presenti nel suo input.

1.7 Funzioni

In C, una funzione è equivalente alla subroutine (o funzione) del Fortran ed alla procedura (o funzione) del Pascal. Una funzione è uno strumento che consente di raggruppare diverse operazioni, il cui risultato può essere riutilizzato in seguito, senza che ci si debba preoccupare di come esso sia stato ottenuto dal punto di vista implementativo. Utilizzando funzioni ideate nel modo opportuno, è possibile persino ignorare *come* un determinato lavoro viene eseguito; è infatti sufficiente conoscere *cosa* viene eseguito. Il C rende facile, conveniente ed efficiente l'uso delle funzioni; per questo motivo, spesso vedrete funzioni brevi invocate una sola volta, dichiarate con il preciso intento di rendere più leggibile un particolare segmento di codice.

Fino ad ora abbiamo utilizzato soltanto funzioni già presenti nelle librerie di sistema, come `printf`, `getchar` e `putchar`; ora è giunto il momento di scrivere qualche funzione implementata da noi. Poiché il C non possiede un operatore esponenziale, equivalente all'operatore `**` del Fortran, illustreremo la mecca-nica della definizione di funzione realizzando la funzione `power(m,n)`, che eleva un intero `m` alla potenza intera positiva `n`. Cioè, il valore di `power(2,5)` è 32. Questa funzione non è particolarmente pratica, perché gestisce soltanto potenze positive di piccoli interi, ma possiede tutte le caratteristiche necessarie alla nostra presentazione (la libreria standard contiene una funzione `pow(x,y)` che calcola `x^y`).

Presentiamo dunque la funzione `power`, insieme ad un programma principale che la utilizza, in modo che possiate osservarne l'intera struttura.

```
#include  <stdio.h>

int power(int m, int n);

/* usa la funzione power */
main()
{
    int i;

    for (i=0; i<10; ++1)
         printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0
}

/* power: eleva base all'n-esima potenza; n >= 0 */
int power(int base, int n)
{
    int i, p;

    p=1;
    for (i=1; i<= n; ++i)
         p=p*base;
    return p;
}
```

Una definizione di funzione ha il seguente formato:

```
tipo_ritornato nome_funzione(lista parametri, se esiste)
{
    dichiarazioni
    istruzioni
}
```

Le definizioni di funzione possono comparire in un ordine qualsiasi, all'interno di uno o più file sorgente, ma nessuna funzione può essere spezzata su file separati. Se il programma sorgente è suddiviso in più file, per

poterlo compilare ed eseguire sono necessarie alcune informazioni aggiuntive, che dipendono però dal sistema operativo e non dal linguaggio. Per il momento, assumiamo che entrambe le funzioni si trovino nello stesso file, in modo che tutto ciò che è stato detto fino ad ora sull'esecuzione dei programmi C rimanga valido.

La funzione `power` viene chiamata due volte da `main`, nella riga

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Ogni chiamata fornisce due argomenti, e la funzione `power` produce, come risultato, un intero che dev'essere formattato e stampato. All'interno di un'espressione, `power(2,i)` è un intero, esattamente come lo sono `2` e `i` (non tutte le funzioni producono un valore intero, ma di queste parleremo nel Capitolo 4).

La prima linea della funzione `power`

```
int  power(int base, int n)
```

dichiara i nomi ed i tipi dei parametri, oltre che il tipo del risultato restituito dalla funzione. I nomi usati da `power` per i suoi parametri sono locali a `power` stessa, e non sono visibili a nessun'altra funzione: le altre routine possono usarli, senza incorrere in problemi di conflitto. Ciò resta vero anche per le variabili `i` e `p`: la `i` utilizzata in `power` non ha alcuna relazione con la `i` usata nel `main`.

Normalmente, noi utilizzeremo il termine *parametro* per una variabile definita nella lista fra parentesi che compare nella definizione di funzione, mentre il termine *argomento* sarà il valore usato in una chiamata del-la funzione stessa. A volte, questo stesso criterio distingue i termini *argomento formale* ed *argomento reale*.

Il valore calcolato da `power` viene restituito al `main` attraverso l'istruzione `return`, che può essere seguita da qualsiasi espressione:

```
return espressione;
```

Una funzione non ha bisogno di restituire un valore; un'istruzione `return` senza alcuna espressione fa sì che al chiamante venga restituito il controllo, ma non un valore utile. È come se la funzione terminasse per il raggiungimento della sua ultima parentesi destra, quella di chiusura.

È da notare che anche al termine del `main` è presente un'istruzione di `return`. Poiché, infatti, il `main` è una funzione analoga a qualsiasi altra, anch'esso può restituire un particolare valore al chiamante, che in realtà è l'ambiente nel quale il programma è stato eseguito. Di solito, un valore di ritorno nullo indica un cor-retto completamento; valori diversi da zero segnalano, invece, la presenza di condizioni di terminazione eccezionali o scorrette. Per semplicità, fino a questo momento avevamo omesso dalle nostre funzioni `main` le istruzioni di `return`; d'ora in poi, tuttavia, le includeremo, per ricordare che ogni programma dovrebbe restituire all'ambiente chiamante uno stato di terminazione.

La dichiarazione

```
int  power(int m, int n);
```

posta appena prima del `main` segnala che `power` è una funzione che si aspetta due argomenti interi e che restituisce un intero. Questa dichiarazione, che viene detta *prototipo della funzione*, deve essere in accordo con la definizione e l'uso di `power`. Il fatto che la definizione o una qualsiasi delle chiamate a `power` non concordino con questo prototipo costituisce un errore.

Questa corrispondenza non è invece richiesta per i nomi dei parametri. Infatti, all'interno del prototipo essi sono addirittura opzionali; quindi, noi avremmo potuto scrivere

```
int  power(int, int);
```

Tuttavia i nomi, se scelti opportunamente, costituiscono una buona documentazione, quindi noi li utilizzeremo spesso.

Una nota storica: la variazione principale dell'ANSI C rispetto alle versioni precedenti riguarda la dichiarazione e la definizione delle funzioni. Nella definizione originaria di una funzione C, la funzione `power` sarebbe apparsa nella forma seguente:

```
/* power: eleva base all'n-esima potenza; n >= 0 */
/*            (versione originaria) */
power(base,n)
int base, n;
{
    int i, p;

    p=1;
    for (i=0; i<=n; ++i)
         p=p*base;
    return p;
}
```

I parametri venivano nominati all'interno delle parentesi, mentre i loro tipi erano dichiarati prima delle parentesi sinistra di apertura; i parametri non dichiarati erano assunti essere `int` (il corpo della funzione non presentava, invece, variazioni di rispetto alla sintassi attuale). La dichiarazione di `power` all'inizio del programma sarebbe stata la seguente:

```
int  power();
```

Con la vecchia sintassi non poteva essere fornita alcuna lista di parametri, cosicché il compilatore non era in grado di controllare subito se `power` era stata chiamata correttamente. Inoltre poiché, per default, si assumeva che `power` restituisse un intero, l'intera dichiarazione avrebbe potuto essere tralasciata.

La nuova sintassi dei prototipi di funzione facilita il compilatore nella ricerca di errori relativi al numero ed al tipo degli argomenti. Il vecchio stile di dichiarazione e definizione sarà ancora supportato dall'ANSI C, almeno per un certo periodo, ma vi invitiamo ad utilizzare la nuova forma ogni volta che possedete un compilatore che la supporta.

Esercizio 1.15 Riscrivete il programma di conversione delle temperature della Sezione 1.2 in modo da usare una funzione di conversione.

1.8 Argomenti - Chiamata per Valore

Per coloro che conoscono ed utilizzano il Fortran, le funzioni C presentano un aspetto poco familiare. In C, tutti gli argomenti delle funzioni vengono passati "per valore". Questo significa che i valori degli argomenti vengono forniti alla funzione in variabili temporanee, piuttosto che in quelle d'origine. Da ciò derivano alcu-ne differenze rispetto ai linguaggi che, come il Fortran, possiedono la "chiamata per riferimento" oppure (è il caso del Pascal) permettono l'uso di parametri preceduti da var; in questi casi, infatti, la funzione chiamata accede direttamente all'argomento originale, e non ad una copia locale di esso.

La differenza principale consiste nel fatto che, in C, la funzione chiamata non può alterare direttamente una variabile nella funzione chiamante; essa può modificare soltanto la sua copia, privata e temporanea.

Contrariamente a quanto può sembrare, la chiamata per valore è un vantaggio; essa consente di scrivere programmi più compatti contenenti meno variabili non essenziali, perché i parametri possono essere considerati delle variabili locali inizializzate nel modo opportuno all'interno della funzione chiamata. Come esempio, presentiamo una versione di `power` che utilizza questa proprietà.

```
/* power: eleva base all'n-esima potenza; n >= 0; versione 2 */
int  power(int base, int n)
{
    int p;

    for (p=1; n>0; --n)
         p=p*base;
    return p;
}
```

Il parametro `n` viene usato come variabile temporanea, e viene decrementato (con un ciclo di `for` che procede a ritroso) fino a che diventa zero; la variabile `i` non è più necessaria. Le modifiche apportate ad `n` allo interno di `power` non hanno effetto sull'argomento con cui `power` era stata originariamente chiamata.

In caso di necessità, è possibile fare in modo che una funzione modifichi una variabile all'interno della routine chiamante. Il chiamante deve fornire l'*indirizzo* della variabile (tecnicamente, il *puntatore*), mentre la funzione chiamata deve dichiarare il parametro come un puntatore attraverso il quale accedere, indirettamente, alla variabile stessa. Illustreremo questo argomento nel Capitolo 5.

Il caso dei vettori è però diverso. Quando il nome di un vettore è usato come argomento, il valore passato alla funzione è la posizione (o indirizzo) dell'inizio del vettore stesso: non viene copiato alcun elemento. Indicizzando questo valore, la funzione può usare e modificare qualsiasi elemento del vettore. Questo argomento è l'oggetto del prossimo paragrafo.

1.9 Vettori di Caratteri

In C, il tipo di vettore più comune è il vettore di caratteri. Per illustrare l'uso degli array di caratteri e delle funzioni che li manipolano, scriviamo un programma che legge un certo insieme di linee di testo e stampa la più lunga. Lo schema è abbastanza semplice:

```
while (c'è un'altra linea)
    if (essa è maggiore della linea più lunga sinora trovata)
         salvala
         salva la sua lunghezza
stampa la linea più lunga
```

Questo schema evidenzia il fatto che il programma può facilmente essere suddiviso in più parti. Una parte legge la linea, un'altra la controlla, una terza la registra, ed una quarta controlla il processo.

Poiché esiste questa suddivisione quasi naturale, sarebbe bene che essa venisse mantenuta anche nella stesura del programma. Quindi, scriviamo innanzitutto una funzione `getline` che legge la prossima linea in input. Cercheremo di utilizzare questa funzione anche in altri contesti. Come requisito minimo, `getline` deve restituire un segnale particolare in caso di fine del testo; un'implementazione più sofisticata potrebbe restituire la lunghezza della linea, oppure zero in caso di terminazione dell'input. Notiamo che zero, in caso di fine del testo, è un valore di ritorno non ambiguo, perché non possono esistere righe di input di lunghezza zero. Ogni linea, infatti, è costituita da almeno un carattere; anche una linea contenente solo un new line ha lunghezza 1.

Quando troviamo una linea più lunga della linea maggiore finora incontrata, la salviamo in qualche area particolare. Questo suggerisce la necessità di una seconda funzione, `copy`, per copiare la nuova linea in un buffer di salvataggio.

Infine, abbiamo bisogno di un programma principale che controlli `getline` e `copy`. Il risultato finale è il seguente:

```
#include  <stdio.h>
#define  MAXLINE  1000  /* lunghezza massima di una linea */

int  getline(char line[], int maxline);
void  copy(char to[], char from[]);

/* stampa la linea di input più lunga */
main()
{
    int  len;                  /* lunghezza della linea corrente */
    int  max;                  /* massima lunghezza trovata sinora */
    char  line[MAXLINE];       /* linea di input corrente */
    char  longest[MAXLINE];    /* linea più lunga salvata qui */

    max=0;
    while ((len=getline(line, MAXLINE))>0)
        if (len>max)
        {
```

```
            max=len;
            copy(longest, line);
        }
    if (max>0)     /* cerca almeno una linea in input */
        printf("%s", longest);
    return 0;
}

/* getline: legge e carica in s una linea, ritorna la lunghezza */
int  getline(char s[], int lim)
{
    int c, i;

    for (i=0; i<line-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i]=c;
    if (c=='\n')
    {
        s[i]=c;
        ++i;
    }

    s[i]='\0';
    return i;
}

/* copy: copia 'from' in 'to'; assume che 'to' sia
sufficientemente ampio */
void  copy(char to[], char from[])
{
    int i;

    i=0;
    while ((to[i]=from[i])!='\0')
        ++i;
}
```

Le funzioni `getline` e `copy` sono dichiarate all'inizio del programma, che supponiamo essere contenuto in un unico file.

`main` e `getline` comunicano tramite una coppia di argomenti ed un valore di ritorno. In `getline`, gli argomenti sono dichiarati nella linea

```
int  getline(char s[], int lim)
```

in cui si specifica che il primo argomento, `s[]`, è un vettore, mentre il secondo, `lim`, è un intero. Normalmente, per potere riservare la memoria necessaria per un array, è necessario specificarne la lunghezza. Nel nostro caso, la lunghezza di `s` non deve essere specificata in `getline`, perché essa è già definita in `main`. `getline`, analogamente a quanto faceva la funzione `power`, utilizza l'istruzione `return` per restitui-re un valore al chiamante. Anche il tipo di questo valore di ritorno, `int`, è specificato in questa linea; poiché questo tipo è quello di default per i valori di ritorno, la sua dichiarazione potrebbe essere tralasciata.

Alcune funzioni restituiscono un valore utile; altre, come `copy`, vengono utilizzate soltanto per i loro effetti, indipendentemente dal valore di ritorno. Il tipo del valore di ritorno di `copy` è `void`, che equivale ad una dichiarazione esplicita del fatto che la funzione non ritorna alcun valore.

Alcune funzioni restituiscono un valore utile; come `copy`, vengono utilizzate soltanto per eseguire le operazioni in esse contenute, senza dover fornire alcun valore alla funzione chiamante. Il tipo del valore di ritorno di `copy` è `void`, che equivale ad una dichiarazione esplicita del fatto che la funzione non restituisce alcun valore.

```
"salve\n"
```

essa viene memorizzata sotto forma di vettore di caratteri, costituito dai caratteri della stringa seguiti da `'\0'`, che ne identifica la fine.

La specifica di formattazione %s, all’interno della printf, si aspetta che l’argomento corrispondente sia una stringa presentata in questa forma. Anche copy si basa sul fatto che il suo argomento di input termini con ‘\0’, e copia questo carattere nell’argomento di output (tutto questo implica che ‘\0’ non possa far parte di un testo normale).

Per inciso, è utile ricordare che anche un programma piccolo come questo presenta alcuni problemi che andrebbero risolti in fase di progetto. Per esempio, cosa dovrebbe fare il main, se incontrasse una linea più lunga del limite massimo definito? getline si comporterebbe correttamente, perché essa termina quando il vettore è pieno, indipendentemente dal fatto che sia stato incontrato o meno un new line. Controllando la lunghezza ed il carattere di ritorno, main potrebbe determinare se la linea era troppo lunga e comportarsi di conseguenza. Per brevità, noi abbiamo ignorato questo aspetto.

Poiché chi invoca getline non può sapere a priori quanto possa essere lunga una linea di input, getline stessa gestisce il caso di overflow. D’altro canto, chi usa copy conosce già, o almeno può ricavare, le dimensioni delle stringhe; di conseguenza, abbiamo scelto di non dotare questa funzione di alcun controllo sugli errori.

Esercizio 1.16 Rivedete il main del programma della “linea più lunga”, in modo che stampi la lunghezza di una stringa arbitrariamente grande, oltre alla massima quantità possibile dei caratteri che la compongono.

Esercizio 1.17 Scrivete un programma per stampare tutte le linee di input che superano gli 80 caratteri.

Esercizio 1.18 Scrivete un programma che rimuova gli spazi ed i caratteri di tabulazione al termine di ogni linea, e che cancelli completamente le linee bianche.

Esercizio 1.19 Scrivete una funzione reverse(s), che inverta la stringa di caratteri s. Usatela per scrive-re un programma che inverta, una per volta, le sue linee di input.

1.10 Variabili Esterne e Scope

Le variabili in main, come line e longest, sono private (o locali) a main. Poiché esse sono dichiarate all’interno di main, nessun’altra funzione può accedervi direttamente. La stessa cosa vale per le variabili dichiarate nelle altre funzioni; per esempio, la variabile i in getline non ha alcuna relazione con la variabile i in copy. Ogni variabile locale ad una funzione viene realmente creata al momento della chiamata alla funzione stessa, e cessa di esistere quando quest’ultima termina. Per questo motivo, le variabili locali sono dette anche variabili *automatiche*. D’ora in poi, per indicare le variabili locali, noi useremo il termine “automatico” (il Capitolo 4 illustra il concetto di static, grazie al quale una variabile locale può conservare il suo valore anche fra una chiamata e l’altra della funzione a cui appartiene).

Poiché le variabili automatiche nascono e muoiono con le chiamate alla funzione, esse non possono mantenere il loro valore fra due riferimenti successivi, e devono quindi essere inizializzate opportunamente ad ogni chiamata. Se ciò non viene fatto, il loro contenuto è “sporco”.

In alternativa alle variabili automatiche, è possibile definire delle variabili *esterne* a qualsiasi funzione, cioè variabili che, tramite il loro nome, possono essere utilizzate da ogni funzione del programma (questo meccanismo è molto simile al COMMON del Fortran ed alle variabili dichiarate nel modulo più esterno di un programma Pascal).

Una variabile esterne sono accessibili globalmente, esse possono sostituire, almeno in parte, le liste di argomenti usate per comunicare i dati da una funzione all’altra. Inoltre queste variabili, non scomparendo al termine dell’esecuzione delle diverse funzioni, sono in grado di conservare il loro valore anche dopo la terminazione della funzione che lo alterato.

Una variabile esterna dev’essere *definita*, una ed una sola volta, al di fuori di qualsiasi funzione; questa operazione implica che venga riservato dello spazio in memoria per questa variabile. La stessa variabile deve anche essere *dichiarata* in ogni funzione che la utilizza; questa dichiarazione stabilisce il tipo della variabile.

La dichiarazione può essere esplicita, con un'istruzione `extern`, oppure implicita, determinata dal contesto. Per rendere più concreta questa discussione, riscriviamo il programma della "linea più lunga" usando `line`, `longest` e `max` come variabili esterne. Questo comporta la necessità di modificare le chiamate, le dichiarazioni ed i corpi di tutte le tre funzioni.

```
#include  <stdio.h>
#define  MAXLINE  1000  /* lunghezza massima di una linea */

int max;                /* massima lunghezza trovata sinora */
char line[MAXLINE];     /* linea di input corrente */
char longest[MAXLINE];  /* linea più lunga salvata qui */

int  getline(void);
char  copy(void);

/* stampa la linea di input più lunga; seconda versione */
main()
{
    int len;
    extern int max;
    extern char longest[MAXLINE];

    max=0;
    while ((len=getline())>0)
        if (len>max)
        {
            max=len;
            copy();
        }
    if (max>0)     /* c'era almeno una linea in input */
        printf("%s", longest);
    return 0;
}

/* getline: seconda versione */
int  getline(void)
{
    int c, i;
    extern char line[];

    for (i=0; i<MAXLINE-1  &&  (c=getchar())!=EOF  &&  c!='\n'; ++i)
        line[i]=c;
    if (c=='\n')
    {
        line[i]=c;
        ++i;
    }
    line[i]='\0';
    return i;
}

/* copy: seconda versione */
void  copy(void)
{
    int i;
    extern char line[], longest[];

    i=0;
    while ((longest[i]=line[i])!='\0')
        ++i;
}
```

Le variabili esterne in `main`, `getline` e `copy` sono definite nelle prime righe, che stabiliscono il loro tipo e consentono di riservare la memoria necessaria alla loro allocazione. Sintatticamente, le definizioni delle variabili esterne sono del tutto analoghe a quelle delle variabili locali. Prima che una funzione possa utilizzare una variabile, il nome di quest'ultima dev'essere reso noto alla funzione stessa. Un modo per farlo consiste

nello scrivere, all'interno della funzione, una dichiarazione di `extern`; tale dichiarazione è uguale a quella precedente, tranne che per il fatto di essere preceduta dalla parola chiave `extern`.

In alcune circostanze, la dichiarazione `extern` in una funzione può essere tralasciata. Questo accade, per esempio, quando la definizione della variabile esterna precede la funzione stessa. Quindi, le dichiarazioni di `extern` presenti in `main`, `getline` e `copy` sono ridondanti. Nella pratica, le definizioni di tutte le variabi-li esterne vengono poste all'inizio del file sorgente, il che consente di omettere tutte le dichiarazioni di `extern`.

Se un programma è distribuito in più file sorgenti, e se una variabile è definita in *file1* ed usata in *file2* e *file3*, allora questi due ultimi file devono contenere una dichiarazione di `extern`, che consenta di correlare le occorrenze della variabile. La prassi consiste nell'inserire le dichiarazioni delle variabili e delle funzioni in un unico file, storicamente chiamato *header*, incluso da ogni file sorgente con un'istruzione di `#include`. Il suffisso convenzionale per gli header è `.h`. Le funzioni della libreria standard, per esempio, sono dichiara-te negli header come `<stdio.h>`. Questo argomento sarà illustrato per esteso nel Capitolo 4, mentre nel Capitolo 7 e nell'Appendice B si parlerà della libreria.

Poiché, nella loro seconda versione, sia `getline` che `copy` non possiedono argomenti, la logica suggerirebbe che i loro prototipi, all'inizio del file, fossero `getline()` e `copy()`. Tuttavia, per compatibilità con i vecchi programmi C, lo standard assume che una lista vuota indichi una dichiarazione del vecchio tipo e, di conseguenza, evita di effettuare tutti i controlli di consistenza; il termine `void`, quindi, dev'essere utilizzato per una lista esplicitamente vuota. Discuteremo ulteriormente questo argomento nel Capitolo 4.

Dovreste notare che stiamo usando con molta attenzione i termini *definizione* e *dichiarazione*. "Definizione" si riferisce ai punti nei quali la variabili viene creata o nei quali le viene riservata della memoria; "dichiara-zione" si riferisce invece ai punti nei quali viene soltanto stabilita la natura della variabile stessa.

Per inciso, notiamo che esiste la tendenza ad usare molto le variabili esterne, perché sembrano semplifica-re le comunicazioni di dati: le liste degli argomenti sono più brevi e le variabili sono a vostra disposizione in qualsiasi punto ed in qualsiasi momento. Ma le variabili esterne sono sempre presenti anche dove e quando voi non le desiderate. Fare un uso troppo massiccio delle variabili esterne è pericoloso, perché conduce alla stesura di programmi nei quali non tutte le connessioni fra i dati sono chiare: le variabili possono essere modificate in punti e in modi inattesi, ed il programma è difficile da modificare. La seconda versione del programma "linea più lunga" è peggiore della prima, in parte per queste ragioni ed in parte perché, inserendo in `getline` ed in `copy` i nomi delle variabili sulle quali operare, distrugge la generalità di due utili funzioni.

A questo punto, abbiamo illustrato quello che può essere definito il nucleo del C. Con questo ristretto insieme di blocchi elementari, è possibile scrivere programmi utili, anche di dimensioni ragguardevoli, e probabilmente sarebbe una buona idea se voi dedicaste un certo tempo a quest'attività. Gli esercizi seguenti suggeriscono programmi di complessità leggermente superiore a quella dei programmi precedentemente descritti in questo capitolo.

Esercizio 1.20 Scrivete un programma `detab` che rimpiazza i caratteri di tabulazione con un numero di spazi tale da raggiungere il successivo tab stop. Assumete di avere un insieme fissato di tab stop, diciamo ogni *n colonne*. *n* dovrebbe essere una variabile o una costante simbolica ?

Esercizio 1.21 Scrivete un programma `entab` che sostituisce le stringhe di caratteri bianchi con il minimo numero di caratteri di tabulazione e di spazi necessari a raggiungere la stessa spaziatura data in input. Usate gli stessi tab stop del programma `detab`. Quando, per raggiungere un tab stop, basta un solo carat-tere di tabulazione od un solo spazio, quale fra i due caratteri è preferibile usare ?

Esercizio 1.22 Scrivete un programma per separare linee di input molto lunghe in due o più linee brevi, spezzando la linea dopo l'ultimo carattere non bianco che cade prima dell'*n*-esima colonna dell'input. Assicuratevi che il vostro programma si comporti ragionevolmente con linee di qualsiasi lunghezza, e che non ci siano caratteri di tabulazione o spazi prima della colonna specificata.

Esercizio 1.23 Scrivete un programma per rimuovere tutti i commenti da un programma C. Non dimenticate di gestire correttamente le stringhe fra apici e le costanti di caratteri. I commenti, in C, non possono essere nidificati.

Esercizio 1.24 Scrivete un programma che faccia dei rudimentali controlli di correttezza sintattica di un programma C, per esempio sul bilanciamento delle parentesi tonde, quadre e graffe. Non dimenticatevi degli

apici, sia singoli che doppi, delle sequenze di escape e dei commenti (questo programma, se fatto in modo generale, risulta molto difficile).

CAPITOLO 2

TIPI, OPERATORI ED ESPRESSIONI

Le variabili e le costanti sono gli oggetti di base utilizzati da un programma. Le dichiarazioni elencano le variabili che dovranno essere usate, e stabiliscono il loro tipo oltre che, eventualmente, il loro valore inizia-le. Gli operatori specificano ciò che deve essere fatto sulle variabili. Le espressioni combinano variabili e costanti per produrre nuovi valori. Il tipo di un oggetto determina l'insieme di valori che esso può assumere e le operazioni che possono essere effettuate su di esso. Questi blocchi elementari costituiscono l'argo-mento di questo capitolo.

Lo standard ANSI ha apportato ai tipi ed alle espressioni di base molti piccoli cambiamenti. Ora, tutti i tipi di interi possono essere `signed` o `unsigned`, ed esistono nuove notazioni per definire costanti prive di segno e costanti esadecimali a caratteri. Le operazioni in floating-point possono essere fatte in singola precisione; per la precisione estesa, esiste il tipo `long double`. Le stringhe costanti possono essere concatenate al momento della compilazione. I tipi enumerativi sono diventati parte del linguaggio, il che ha consentito di formalizzare una prassi già da lungo tempo adottata. Dichiarando un oggetto come `const`, è possibile impedire che esso venga alterato. Le regole per le conversioni automatiche fra tipi aritmetici sono state ampliate, per potere gestire il nuovo e più vasto insieme di possibilità.

2.1 Nomi di Variabili

Nel Capitolo 1 non abbiamo precisato che sui nomi delle variabili e delle costanti esistono alcune restrizioni. I nomi sono costituiti da lettere e da cifre; il primo carattere deve sempre essere una lettera. L'underscore ("_"), che talvolta consente di aumentare la leggibilità di nomi molto lunghi, viene considerato una lettera. Poiché le lettere maiuscole e quelle minuscole sono distinte, x ed X rappresentano due caratteri diversi. La prassi del C adotta lettere minuscole per i nomi delle variabili, mentre per quelli delle costanti impiega esclusivamente caratteri maiuscoli.

I nomi interni sono composti da almeno 31 caratteri significativi. Per i nomi delle funzioni e delle variabili esterne, questo numero può anche essere inferiore, perché può accadere che i compilatori e gli assemblatori usino nomi esterni, sui quali il linguaggio non ha alcun controllo. Per i nomi esterni, lo standard garantisce l'unicità soltanto sui primi 6 caratteri. Le parole chiave come `if`, `else`, `int`, `float`, `...`, sono riservate: non possono, cioè, essere utilizzate come nomi di variabili. Inoltre, esse devono essere scritte a lettere minuscole.

È bene scegliere nomi legati allo scopo delle diverse variabili, e che siano difficilmente confondibili tra loro. In questo libro, tendiamo ad usare nomi brevi per le variabili locali, ed in particolare per gli indici dei cicli, e nomi più lunghi per le variabili esterne.

2.2 Tipi di Dati e Dimensioni

In C esiste un ristretto numero di tipi di dati fondamentali:

`char`	un singolo byte, in grado di rappresentare uno qualsiasi dei caratteri del set locale;
`int`	un intero, che in genere riflette l'ampiezza degli interi sulla macchina utilizzata;
`float`	floating-point in singola precisione;
`double`	floating-point in doppia precisione.

In aggiunta esistono alcuni qualificatori, che possono essere applicati a questi tipi; per esempio, `short` e `long` sono associabili agli interi:

```
short int sh;
long int counter;
```

In queste dichiarazioni il termine `int` può, ed in genere viene, omesso.

Lo scopo è quello di fornire, ove necessario, delle lunghezze diverse per gli interi; normalmente, l'ampiezza di un `int` rispecchia quella degli interi sulla macchina utilizzata. Spesso, `short` indica un intero di 16 bit, e

`long` uno di 32, mentre `int` occupa 16 o 32 bit. Ogni compilatore è libero di scegliere la dimensione appropriata all'hardware sul quale opera, e deve sottostare unicamente alla restrizione secondo la quale gli `short` e gli `int` devono essere di almeno 16 bit, i `long` di almeno 32, e gli `short` non possono essere più lunghi degli `int`, che a loro volta non devono superare i `long`.

I qualificatori `signed` ed `unsigned` possono essere associati sia ai `char` che agli `int`. I numeri `unsigned` sono sempre positivi o nulli, ed obbediscono alle leggi dell'aritmetica modulo 2^n, dove n è il numero di bit del tipo. Per esempio, se i `char` occupano 8 bit, le variabili `unsigned char` vanno da -128 a +127 (in una macchina che lavori in complemento a due). Una variabile di tipo `char` può avere o meno il segno, in base alla macchina sulla quale ci si trova; in ogni caso, tutti i caratteri stampabili sono sempre positivi.

Il tipo `long double` specifica variabili floating-point in doppia precisione. Analogamente a quanto accade per gli interi, anche l'ampiezza degli oggetti di tipo floating-point dipende dall'implementazione; `float`, `double` e `long double` possono rappresentare una, due o tre diverse ampiezze.

Gli header file standard `<limits.h>` e `<float.h>` contengono le costanti simboliche relative a tutte queste ampiezze ed altre proprietà, illustrate nell'Appendice B, legate alla macchina ed al compilatore.

Esercizio 2.1 Scrivete un programma per determinare il range di variabili di tipo `char`, `short`, `int` e `long`, sia `signed` che `unsigned`. Ricavate questi valori stampando le appropriate costanti, lette dagli header, ed eseguendo calcoli appositi. Determinate anche il range dei diversi tipi di variabili floating-point.

2.3 Costanti

Una costante intera, come `1234`, è un `int`. Una costante `long` è seguita da una `'l'` o da una `'L'` terminali, come `123456789L`; un intero troppo grande per essere contenuto in un `int` verrà considerato `long`. Le costanti prive di segno sono terminate da una `'u'` o una `'U'`, mentre i suffissi `'ul'` ed `'UL'` indicano gli `unsigned long`.

Le costanti floating-point contengono il punto decimale (`123.4`) oppure un esponente (`1e-2`), oppure entrambi; il loro tipo, anche se sono prive di suffisso, è sempre `double`. I suffissi `'f'` ed `'F'` indicano una costante `float`, mentre `'l'` ed `'L'` indicano una costante `long double`.

Il valore di un intero può essere specificato in decimale, in ottale o in esadecimale. Uno `0` preposto ad un intero indica la notazione ottale; un prefisso `0x` (o `0X`) indica invece la notazione esadecimale. Per esem-pio, il numero decimale 31 può essere scritto come `037` (in ottale) e come `0x1f` o `0x1F` (in esadecimale). Anche le costanti ottali ed esadecimali possono essere seguite da un suffisso `'L'` che le dichiara di tipo `long` o da un suffisso `'U'` che le dichiara `unsigned`: `OxFUL` è una costante `unsigned long` con valore decimale 15.

Una *costante carattere* è un intero, scritto sotto forma di carattere racchiuso tra apici singoli, come `'x'`. Il valore di una costante carattere è il valore numerico di quel carattere all'interno del set della macchina. Per esempio, nel codice ASCII la costante carattere `'0'` ha valore 48, che non ha nessun legame con il valore numerico `0`. Scrivendo `'0'` invece di 48, che è un valore numerico dipendente dal set di caratteri, il programma risulta indipendente dal valore particolare, oltre che più leggibile. Le costanti carattere possono apparire nelle espressioni numeriche, alla stregua di interi qualsiasi, anche se vengono utilizzate prevalentemente per i confronti con altri caratteri.

Alcuni caratteri, come `\n` (new line), possono essere rappresentati come costanti o stringhe tramite le sequenze di escape le quali, pur apparendo come stringhe di due caratteri, ne rappresentano uno soltanto. Inoltre, è possibile specificare un arbitrario bit-pattern con lunghezza multipla di un byte, usando la forma

```
'\ooo'
```

dove *ooo* è una sequenza, di lunghezza compresa fra uno e tre, di cifre ottali (`0 ... 7`); oppure, si può scrivere

```
'\xhh'
```

dove *hh* indica una o più cifre esadecimali (`0 ... 9`, `a ... f`, `A ... F`). Quindi, potremmo scrivere

```
#define  VTAB     '\013'      /* Tab verticale in ASCII */
#define  BELL     '\007'      /* Campanello in ASCII  */
```

oppure, in esadecimale,

```
#define  VTAB     '\xb'       /* Tab verticale in ASCII */
#define  BELL     '\x7'       /* Campanello in ASCII */
```

L'insieme completo delle sequenze di escape è il seguente:

```
\a          allarme (campanello)
\b          backspace
\f          salto pagina
\n          new line
\r          ritorno carrello (return)
\t          tab orizzontale
\v          tab verticale
\\          backslash
\?          punto interrogativo
\'          apice singolo
\"          doppi apici
\ooo        numero ottale
\xhh        numero esadecimale
```

La costante carattere `'\0'` rappresenta il carattere con valore zero, cioè il carattere nullo. Spesso, invece di `0`, si scrive `'\0'` per enfatizzare la natura alfanumerica di una particolare espressione, anche se il valore rappresentato è 0.

Un'*espressione costante* è un'espressione costituita da sole costanti. Le espressioni di questo tipo possono essere valutate al momento della compilazione, invece che run-time, e, quindi, possono essere inserite in ogni posizione nella quale può trovarsi una costante; per esempio:

```
#define  MAXLINE  1000
char line[MAXLINE+1];
```

oppure

```
#define  LEAP  1        /* in anni bisestili */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

Una *stringa costante*, o *costante alfanumerica*, è una sequenza di zero o più caratteri racchiusi fra doppi apici, come

```
"Io sono una stringa"
```

oppure

```
""          /* la stringa vuota */
```

Gli apici non fanno parte della stringa: essi servono unicamente a delimitarla. Le stesse sequenze di esca-pe usate per le costanti carattere possono essere inserite nelle stringhe; `\"` rappresenta il carattere di dop-pio apice. Le stringhe costanti possono essere concatenate al momento della compilazione:

```
"Salve,"    " mondo"
```

equivale a

```
"Salve, mondo"
```

Questo meccanismo risulta utile perché consente di spezzare su più linee le stringhe molto lunghe.

Da un punto di vista tecnico, una stringa costante è un vettore di caratteri. Poiché, nella sua rappresenta-zione interna, ogni stringa è terminata dal carattere nullo `'\0'`, la memoria fisica richiesta per ogni stringa è maggiore di un'unità rispetto al numero dei caratteri racchiusi fra apici. Questa rappresentazione consente di non porre limiti superiori alla lunghezza delle stringhe, per determinare la quale ogni programma può

scandire completamente la stringa in esame. La funzione `strlen(s)`, della libreria standard, restituisce la lunghezza del suo argomento `s`, che è una stringa di caratteri, escludendo automaticamente il carattere di terminazione `'\0'`. Presentiamo qui la nostra versione di questa funzione:

```
/* strlen: ritorna la lunghezza di s */
int strlen(char s[])
{
    int i;

    i=0;
    while (s[i]!='\0')
        ++i;
    return i;
}
```

Nell'header file standard `<string.h>` sono definite `strlen` ed altre funzioni per il trattamento delle stringhe.

Fate molta attenzione nel distinguere una costante carattere da una stringa che contiene un singolo carattere: `'x'` e `"x"` non sono la stessa cosa. `'x'` è un intero, usato per produrre il valore numerico della lettera *x* nel set di caratteri della macchina; `"x"`, invece, è un vettore di caratteri che contiene un carattere (la lettera *x*) ed uno `'\0'`.

Esiste, infine, un altro tipo di costanti: le *costanti enumerative*. Un'enumerazione è una lista di valori interi costanti, come

```
enum  boolean  { NO, YES };
```

Il primo nome, in una costante `enum`, ha valore 0, il secondo 1, e così via, a meno che non vengono specificati dei valori espliciti. Se non tutti i valori sono specificati, quelli non specificati continuano la progressione a partire dall'ultimo specificato, come nel secondo di questi esempi:

```
enum  escape  { BELL='\a', BACKSPACE='\b', TAB='\t',
                NEWLINE='\n', VTAB='\v', RETURN='\r' };
enum  mesi  { GEN=1, FEB, MAR, APR, MAG, GIU, LUG, AGO,
              SETT, OTT, NOV, DIC };
      /* FEB è 2, MAR è 3 e così via */
```

I nomi in enumerazioni diverse devono essere distinti. All'interno di un'unica enumerazione, i valori non devono essere necessariamente distinti.

Le enumerazioni sono un mezzo efficiente per associare valori costanti a dei nomi; esse sono, cioè, un'alternativa alla `#define`. Sebbene sia possibile dichiarare variabili di tipo `enum`, i compilatori non controllano che il contenuto di queste variabili sia significativo per l'enumerazione. Ciò nondimeno, le variabili enumerative offrono la possibilità di essere controllate e, per questo motivo, sono spesso preferibili alle `#define`. Va notato, infine, che un debugger potrebbe anche essere in grado di stampare nella loro forma simbolica i valori di una variabile di tipo enumerativo.

2.4 Dichiarazioni

Tutte le variabili, prima di essere utilizzate, devono essere state dichiarate, anche se alcune dichiarazioni possono essere fatte in modo implicito nel contesto. Una dichiarazione specifica un tipo, e contiene una lista di una o più variabili di quel tipo, come

```
int lower, upper, step;
char c, line[1000];
```

Le variabili possono essere distribuite in modi diversi; le liste dell'esempio precedente sono equivalenti a

```
int lower;
int upper;
int step;
char c;
```

```
char line[1000];
```

Quest'ultima forma occupa uno spazio maggiore, ma è utile se si devono aggiungere commenti ad ogni dichiarazione o se si devono apportare modifiche successive.

Nella dichiarazione, una variabile può anche essere inizializzata. Se il nome è seguito dall'operatore = e da un'espressione, allora quest'ultima è l'entità inizializzante, come in

```
char esc='\\';
int i=0;
int limit=MAXLINE+1;
float eps=1.0e-5;
```

Se la variabile non è automatica, l'inizializzazione viene fatta una volta sola, concettualmente prima dell'ini-zio dell'esecuzione del programma, e l'entità inizializzante dev'essere un'espressione costante. Una varia-bile automatica inizializzata esplicitamente viene inizializzata ogni volta che si entra nella funzione o, co-munque, nel blocco che la contiene; in questo caso, l'entità inizializzante può essere un'espressione qual-siasi. Le variabili esterne e quelle statiche sono inizializzate, per default, a zero. Le variabili automatiche non esplicitamente inizializzate contengono valori indefiniti.

Il qualificatore `const` può essere applicato alla dichiarazione di qualsiasi variabile, per specificare che quel valore non verrà mai alterato. Per un vettore, il qualificatore `const` dichiara che nessun elemento del vetto-re verrà modificato.

```
const double e=2.71828182845905;
const char msg[]="warning";
```

La dichiarazione `const` può essere usata anche per i vettori passati come argomenti, per indicare che la funzione chiamata non altera il vettore:

```
int  strlen(const char[]);
```

Se si tenta di modificare un oggetto dichiarato `const`, il risultato dipende dall'implementazione adottata sul-la macchina in uso.

2.5 Operatori Aritmetici

Gli operatori aritmetici binari sono `+`, `-`, `*`, `/`, e l'operatore modulo `%`. La divisione intera tronca qualsiasi parte frazionaria. L'espressione

```
x % y
```

ritorna il resto della divisione di `x` per `y`, e quindi restituisce 0 se `x` è un multiplo di `y`. Per esempio, un anno è bisestile se è divisibile per 4 ma non per 100, fatta eccezione per gli anni divisibili per 400, che *sono* bise-stili. Quindi

```
if ((year%4==0  &&  year%100!=0)  ||  year%400==0)
    printf("Il %d è un anno bisestile\n", year);
else
    printf("Il %d non è un anno bisestile\n", year);
```

L'operatore `%` non è applicabile ai `float` ed ai `double`. Per i numeri negativi, la direzione di troncamento per `/` ed il segno del risultato di `%` dipendono dalla macchina, analogamente a quanto accade per le azioni intraprese in caso di overflow o di underflow.
Gli operatori binari `+` e `-` hanno la stessa precedenza, che è inferiore a quella di `*`, `/` e `%`, a loro volta aventi precedenza inferiore agli operatori unari `+` e `-`. Tutti gli operatori aritmetici sono associativi da sinistra a destra.

La Tabella 2.1, al termine di questo capitolo, riassume le precedenze e le associatività per tutti gli operatori.

2.6 Operatori Relazionali e Logici

Gli operatori relazionali sono

```
>     >=    <     <=
```

Essi hanno tutti la stessa precedenza. Immediatamente sotto di loro, nella scala delle precedenze, trovia-mo gli operatori di uguaglianza:

```
==    !=
```

Gli operatori relazionali hanno precedenza inferiore agli operatori aritmetici, perciò un'espressione come `i<lim-1` viene interpretata come `i<(lim-1)`.

Più interessanti sono gli operatori logici `&&` e `||`. Le espressioni connesse da `&&` e `||` vengono valutate da sinistra a destra, e la valutazione si blocca non appena si determina la verità o la falsità dell'intera espressione. La maggior parte dei programmi C fa affidamento su queste proprietà. Per esempio, riportiamo di seguito il ciclo `for` presente nella funzione `getline` che avevamo implementato nel Capitolo 1:

```
for (i=0; i<lim-1  &&  (c=getchar())!='\n'  &&  c!=EOF; ++i)
    s[i]=c;
```

Prima di leggere un nuovo carattere, è necessario controllare che ci sia spazio sufficiente per memorizzarlo nel vettore `s`, perciò il test `i<lim-1` *deve* essere effettuato per primo. Infatti, se questo test fallisce, noi non dobbiamo proseguire nella lettura dei caratteri di input. Analogamente, confrontare `c` con `EOF` prima di aver eseguito `getchar` sarebbe controproducente; quindi la chiamata a `getchar` e l'assegnamento del suo risultato a `c` devono precedere i controlli su `c` stesso.

La precedenza di `&&` è superiore a quella di `||`, ed entrambi hanno precedenza inferiore a quella degli operatori relazionali e di uguaglianza, quindi espressioni come

```
i<lim-1  &&  (c=getchar())!='\n'  && c!=EOF
```

non necessitano di altre parentesi. Ma poiché la precedenza di `!=` è superiore a quella dell'assegnamento, le parentesi sono necessarie in

```
(c=getchar())!='\n'
```

per ottenere il risultato desiderato, che consiste nell'assegnare a `c` il carattere letto da `getchar` e confrontare quindi `c` con `'\n'`.

Per definizione, il valore numerico di un'espressione logica o relazionale è 1 se la relazione è vera. 0 se essa è falsa.

L'operatore unario di negazione, `!`, converte un operando non nullo in uno 0, ed un operando nullo in un 1. Un uso molto comune dell'operatore `!` è il seguente

```
if (!valid)
```

al posto di

```
if (valid==0)
```

È molto difficile dire quale sia, in generale, la forma migliore. Costruzioni come `!valid`, infatti, sono facilmente leggibili ("se valid è falso"), ma espressioni più complesse possono risultare poco comprensibili.

Esercizio 2.2 Scrivete un ciclo equivalente al ciclo di `for` presentato sopra, senza utilizzare `&&` e `||`.

2.7 Conversioni di Tipo

Quando un operatore ha operandi di tipo diverso, questi vengono convertiti in un tipo comune, secondo un ristretto insieme di regole. In generale, le uniche conversioni automatiche sono quelle che trasformano un operando "più piccolo" in uno "più grande" senza perdita di informazione, come nel caso della conversione di un intero in un floating-point, in espressioni del tipo `f+i`. Espressioni prive di senso, come per esempio l'uso

di un `float` come indice, non sono consentite. Espressioni che possono provocare una perdita di informazione, come l'assegnamento di un intero ad un short o quello di un float ad un intero, producono al più un messaggio di warning, ma non sono illegali.

Poiché un `char` non è altro che un piccolo intero, i `char` possono essere usati liberamente in qualsiasi espressione aritmetica. Questo consente di ottenere una notevole flessibilità in alcuni tipi di trasformazioni di caratteri. Un esempio di ciò è dato da questa semplice implementazione della funzione `atoi`, che converte una stringa di cifre nel suo equivalente numerico.

```
/* atoi: converte s in un intero */
int  atoi(char s[])
{
    int i, n;

    n=0;
    for (i=0; s[i]>='0'  &&  s[i]<='9'; ++i)
        n=10*n+s[i]-'0';
    return n;
}
```

Come abbiamo detto nel Capitolo 1, l'espressione

```
s[i]-'0'
```

fornisce il valore numerico del carattere memorizzato in `s[i]`, perché i valori di `'0'`, `'1'`, ecc., formano una sequenza contigua crescente.

Un altro esempio di conversione di `char` in `int` è dato dalla funzione `lower`, che traduce un singolo carattere nel suo corrispondente minuscolo appartenente al *set di caratteri ASCII*. Se il carattere in input non è maiuscolo, `lower` lo restituisce inalterato.

```
/* lower: converte c in minuscolo; solo ASCII */
int  lower(int c)
{
    if (c>='A'  &&  c<='Z')
        return c+'a'-'A';
    else
        return c;
}
```

Questa funzione è corretta per il codice ASCII, perché in esso ogni lettera maiuscola è separata dal suo corrispettivo minuscolo da una distanza numerica conosciuta, ed ogni alfabeto (maiuscolo e minuscolo) è contiguo: tra `A` e `Z` non esistono caratteri che non siano lettere. Quest'ultima osservazione non è valida per il set di caratteri EBCDIC, sul quale la funzione `lower` non opera correttamente.

L'header standard `<ctype.h>`, descritto nell'Appendice B, definisce una famiglia di funzioni che forniscono meccanismi di controllo e conversione indipendenti dal set di caratteri. Per esempio, la funzione `tolower(c)` restituisce il valore minuscolo di `c` se `c` è maiuscolo, quindi, `tolower` è un sostituto, portabile, della funzione `lower` sopra descritta. Analogamente, il controllo

```
c>='0'  &&  c<='9'
```

può essere sostituito con

```
isdigit(c)
```

D'ora in poi, useremo spesso le funzioni definite in `<ctype.h>`.

Per quanto concerne la conversione dei caratteri in interi, è necessario fare una sottile osservazione. Il linguaggio non specifica se le variabili di tipo `char` siano oggetti con o senza segno. Quando un `char` viene convertito in un `int`, il risultato può essere negativo. La risposta varia da macchina a macchina, in base alle differenze architetturali. Su alcuni sistemi, un `char` il cui bit all'estrema sinistra è pari ad 1 viene convertito in

un intero negativo ("estensione del segno"). Su altri, un `char` viene trasformato in un `int` aggiungendo degli zeri alla sua sinistra, il che produce sempre interi positivi o nulli.

La definizione del C garantisce che nessun carattere che appartiene all'insieme standard di caratteri stampabili della macchina diventi mai negativo; quindi tali caratteri, nelle espressioni, avranno sempre valori positivi. Al contrario, bit-pattern arbitrari memorizzati in variabili di tipo carattere possono risultare negativi su alcune macchine e positivi su altre. Per motivi di portabilità, se dovete memorizzare dati che non siano caratteri in variabili di tipo `char`, specificate sempre se sono `signed` o `unsigned`.

Le espressioni relazionali, come `i<j`, e quelle logiche connesse da `&&` e da `||` valgono, per definizione, 0 se sono false ed 1 se sono vere. Perciò l'assegnamento

```
d = c>='0'  && c<='9'
```

assegna a `d` il valore 1 se `c` è una cifra, 0 altrimenti. Tuttavia, funzioni come `isdigit`, quando sono vere, possono restituire un qualsiasi valore non nullo. Nella parte di controllo di costrutti come `if`, `while`, `for`, ecc., il "vero" significa semplicemente "non nullo", quindi un comportamento simile a quello di `isdigit` non produce problemi.

Le conversioni aritmetiche implicite operano secondo criteri intuitivi. In generale, se un operatore binario come `+` o `*` ha operandi di tipo diverso, il tipo "inferiore" viene *trasformato* nel tipo "superiore" prima di effettuare l'operazione. Il risultato, quindi, appartiene al tipo "superiore". La Sezione 6 dell'Appendice A stabilisce con precisione le regole di conversione. Se non ci sono operandi `unsigned`, tuttavia, il seguente insieme informale di regole è sufficiente per gestire tutti i casi possibili:

> Se c'è un operando `long double`, l'altro viene convertito in un `long double`.
> In caso contrario, se c'è un operando `double`, l'altro viene convertito in un `double`.
> Altrimenti, se c'è un operando `float`, l'altro viene convertito in un `float`.
> Altrimenti, i `char` e gli `short` vengono convertiti in `int`.
> Infine, se c'è un operando `long`, l'altro viene convertito in un `long`.

Notiamo che, in un'espressione, i `float` non vengono convertiti automaticamente in `double`; questa è una differenza rispetto alla definizione originale del C. In generale, le funzioni matematiche come quelle definite in `<math.h>` usano la doppia precisione. La ragione principale dell'uso dei `float` risiede nella necessità di risparmiare memoria nei grandi vettori e, più raramente, nel risparmio di tempo ottenibile su macchine per le quali l'aritmetica in doppia precisione è molto costosa.

Le regole di conversione si complicano in presenza di operandi `unsigned`. Il problema è dato dal fatto che il risultato dei confronti fra oggetti con e senza segno dipende dalla macchina, perché è legato alle ampiezze dei diversi tipi di interi. Per esempio, supponiamo che gli `int` occupino 16 bit, ed i `long` 32. Allora `-1L<1U`, perché `1U`, che è un `int`, viene trasformato in un `signed long`. Ma `-1L<1UL`, perché `-1l` viene convertito in un `unsigned long`, e diventa quindi un numero positivo molto grande.

Anche sugli assegnamenti vengono effettuate delle conversioni; il valore del lato destro viene trasformato nel tipo del valore di sinistra, che è anche il tipo del risultato.

Un carattere viene convertito in un intero, con o senza l'estensione del segno, secondo quanto si è detto sopra.

Gli interi vengono convertiti in `short` o in `char` sopprimendo i loro bit più significativi in eccesso. Perciò, in

```
int i;
char c;

i=c;
c=i;
```

Il valore di `c` rimane inalterato. Questo è vero indipendentemente dal fatto che venga applicata o meno la estensione del segno. Tuttavia, invertendo l'ordine degli assegnamenti si potrebbe verificare una perdita di informazione.

Se x è un float ed i è un int, allora sia x=i che i=x provocano una conversione; il passaggio da float ad int, però, comporta il troncamento della parte frazionaria. Quando un double viene convertito in un float, il fatto che il suo valore venga troncato piuttosto che arrotondato dipende dalla macchina.

Poiché l'argomento di una chiamata di funzione è un'espressione, le conversioni di tipo hanno luogo anche quando ad una funzione vengono passati degli argomenti. In assenza di un prototipo della funzione, i char e gli short vengono convertiti in int, ed i float diventano dei double. Questo è il motivo per cui abbiamo dichiarato gli argomenti di una funzione int e double, anche quando la funzione viene chiamata con degli argomenti char e float.

Infine, in qualsiasi espressione è possibile forzare particolari conversioni, tramite un operatore unario detto *cast*. Nella costruzione

```
(nome_del_tipo) espressione
```

l'*espressione* viene convertita nel tipo specificato, secondo le regole sin qui descritte. Un cast equivale ad assegnare l'*espressione* ad una variabile del tipo specificato, che viene poi utilizzata al posto dell'intera costruzione. Per esempio, la routine di libreria sqrt si aspetta come argomento un double e, se si trova ad operare su qualcosa di diverso, produce un risultato privo di significato (sqrt è dichiarata in <math.h>). Quindi, se n è un intero, possiamo scrivere

```
sqrt((double) n)
```

per convertire il valore di n in un double prima di passarlo a sqrt. Notiamo che il cast produce il *valore* di n nel tipo corretto: n rimane inalterato. L'operatore cast ha la stessa elevata precedenza degli altri operatori unari, secondo quanto riassunto nella tabella presentata al termine di questo capitolo.

Se gli argomenti sono dichiarati dal prototipo della funzione, come normalmente dovrebbe accadere, ad ogni chiamata della funzione la dichiarazione provoca la conversione automatica di tutti gli argomenti. Quindi, con il prototipo

```
double sqrt(double);
```

per la funzione sqrt, la chiamata

```
root2=sqrt(2);
```

forza l'intero 2 nel valore double 2.0 senza che sia necessario alcun cast.

La libreria standard include un'implementazione portabile di un generatore di numeri pseudo-casuali ed una funzione per inizializzarne il seme; la prima di queste due funzioni include un cast:

```
unsigned long int next=1;

/* rand: ritorna un numero pseudo-casuale
   compreso tra 0 e 32767 */
int  rand(void)
{
    next = next*1103515245+12345;
    return (unsigned int)(next/65536)%32768;
}

/* srand: inizializza il seme per rand() */
void  srand(unsigned int seed)
{
    next=seed;
}
```

Esercizio 2.3 Scrivete la funzione htoi(s), che converte una stringa di cifre esadecimali (che può comprendere un suffisso opzionale 0x o 0X) nel suo valore intero corrispondente. Le cifre disponibili vanno da 0 a 9, da a ad f e da A ad F.

2.8 Operatori di Incremento e Decremento

Per incrementare e decrementare le variabili, il C offre due operatori alquanto insoliti. L'operatore di incremento ++ aggiunge 1 al suo operando, mentre l'operatore di decremento – sottrae 1. Nei paragrafi precedenti, noi abbiamo usato spesso l'operatore ++ per incrementare le variabili, come in

```
if (c=='\n')
    ++nl;
```

La caratteristica insolita è data dal fatto che ++ e – possono essere utilizzati in notazione sia prefissa (cioè prima della variabile, come in ++n) che postfissa (cioè dopo la variabile, come in n++). In entrambi i casi, lo effetto è l'incremento della variabile n. Tuttavia, l'espressione ++n incrementa n *prima* di utilizzarne il valore, mentre n++ incrementa n *dopo* che il suo valore è stato utilizzato. Questo significa che, in un contesto in cui viene utilizzato il valore di n, ++n ed n++ hanno significati diversi, anche se entrambe le espressioni producono un incremento di n. Se n vale 5, allora

```
x=n++;
```

assegna a x il valore 5, ma

```
x=++n;
```

assegna a x il valore 6. In entrambi i casi, n diventa 6. Gli operatori di incremento e decremento possono essere applicati soltanto alle variabili; un'espressione come (i+j)++ è illegale.

In un contesto in cui è significativo soltanto l'incremento, come in

```
if (c=='\n')
    ++nl;
```

le notazioni prefissa e postfissa sono del tutto equivalenti. Tuttavia, esistono situazioni nelle quali viene esplicitamente richiesto l'uso di una notazione piuttosto che dell'altra. Per esempio, consideriamo la funzione squeeze(s,c), che rimuove dalla stringa s tutte le occorrenze del carattere c.

```
/* squeeze: elimina da s tutte le occorrenze di c */
void  squeeze(char s[], int c)
{
    int i, j;

    for (i=j=0; s[i]!='\0'; i++)
        if (s[i]!=c)
            s[j++]=s[i];
    s[j]='\0';
}
```

Ogni volta che si incontra un carattere diverso da c, tale carattere viene copiato nella posizione corrente j, e soltanto allora j viene incrementato per disporsi a ricevere il carattere successivo diverso da c. Questo costrutto è del tutto equivalente al seguente:

```
if (s[i]!=c)
{
    s[j]=s[i];
    j++;
}
```

Un altro esempio di una costruzione di questo tipo può essere ricavato dalla funzione getline che abbiamo scritto nel Capitolo 1, all'interno della quale possiamo sostituire

```
if (c=='\n')
{
    s[i]=c;
    ++i;
}
```

con la forma più compatta

```
if (c=='\n')
    s[i++]=c;
```

Come terzo esempio, consideriamo la funzione standard `strcat(s,t)`, che concatena la stringa `t` al termine della stringa `s`. `strcat` assume che in `s` ci sia spazio sufficiente a contenere il risultato della concatenazione. Nella versione scritta da noi, `strcat` non restituisce alcun valore; la versione presente nella libreria standard, invece, ritorna un puntatore alla stringa risultante.

```
/* strcat: concatena t ad s; s deve essere sufficientemente grande */
void  strcat(char s[], char t[])
{
    int i, j;

    i=j=0;
    while (s[i]!='\0')                    /* trova la fine di s */
        i++;
    while ((s[i++]=t[j++])!='\0')         /* copia t */
        ;
}
```

Ogni volta che un carattere di `t` viene copiato in `s`, sia ad `i` che a `j` viene applicato l'operatore postfisso `++`, per assicurarsi che entrambi gli indici siano nella posizione corretta per l'assegnamento successivo.

Esercizio 2.4 Scrivete una versione alternativa della funzione `squeeze(s1, s2)`, che cancelli da `s1` tutti i caratteri che compaiono anche nella *stringa* `s2`.

Esercizio 2.5 Scrivete la funzione `any(s1, s2)`, che ritorna la prima posizione della stringa `s1` in cui compare uno qualsiasi dei caratteri di `s2`, oppure `-1` se `s1` non contiene alcun carattere di `s2` (la funzione della libreria standard `strpbrk` esegue questo compito, ma restituisce un puntatore alla locazione).

2.9 Operatori Bit a Bit

Per la manipolazione dei bit, il C fornisce sei operatori, applicabili soltanto ad operandi interi, cioè `char`, `short`, `int` e `long`, con o senza segno.

`&`	AND bit a bit
`|`	OR inclusivo bit a bit
`^`	OR esclusivo bit a bit
`<<`	shift a sinistra
`>>`	shift a destra
`~`	complemento a uno (unario)

L'operatore di AND bit a bit `&` viene spesso utilizzato per azzerare particolari insiemi di bit; per esempio

```
n = n & 0177;
```

azzera tutti i bit di `n`, esclusi i 7 meno significativi.
L'operatore di OR bit a bit `|`, invece, viene spesso impiegato per attivare particolari insiemi di bit:

```
x = x | SET_ON;
```

pone a uno, in `x`, i bit che sono pari a uno in `SET_ON`.

L'operatore di OR esclusivo bit a bit `^`, pone ad uno tutti i bit che si trovano in posizioni nelle quali i bit dei due operandi hanno valore diverso, ed azzera quelli per i quali i bit degli operandi sono uguali.

È necessario distinguere gli operatori bit a bit `&` e `|` dagli operatori logici `&&` e `||`, che implicano la valutazione da sinistra a destra di un valore di verità. Per esempio, se `x` vale 1 e `y` vale 2, allora `x&y` vale 0, mentre `x&&y` vale 1.

Gli operatori di shift `<<` e `>>` spostano, verso sinistra e verso destra rispettivamente, il loro operando sinistro di un numero di bit pari al valore del loro operando destro, che dev'essere positivo. Per esempio, `x<<2` spo-

sta a sinistra di due posizioni il valore di x, riempiendo con degli zeri le posizioni così liberatesi; questo equivale a moltiplicare x per 4. Eseguendo un shift a destra di una quantità unsigned, i bit rimasti liberi vengono sempre posti a zero. Eseguendo invece lo shift a destra di una quantità signed, il risultato dipende dalla macchina: su alcuni sistemi, i bit rimasti liberi vengono posti uguali al bit di segno ("shift aritmetico"), mentre su altri vengono posti a zero ("shift logico").

L'operatore unario ~ produce il complemento ad uno di un intero; cioè, esso converte ogni bit attivo in un bit nullo e viceversa. Per esempio,

```
x = x & ~077
```

pone a zero gli ultimi sei bit di x. Notiamo che l'espressione x & ~077 è indipendente dalla lunghezza della parola, ed è quindi preferibile, per esempio, alla forma x & 0177700, nella quale si assume che x sia una quantità di 16 bit. Notiamo anche che la forma portabile non è più costosa, in quanto ~077 è un'espressione costante, che può essere valutata al momento della compilazione.

A chiarimento dell'uso di questi operatori, consideriamo la funzione getbits(x, p, n), che restituisce n bit (allineati a destra) di x a partire dalla posizione p. Assumiamo che il bit 0 sia quello all'estrema destra, e che n e p siano dei numeri positivi significativi. Per esempio, getbits(x, 4, 3) restituisce, allineandoli a destra, i tre bit di x che si trovano nelle posizioni 4, 3, 2.

```
/* getbits: preleva n bit a partire dalla posizione p */
unsigned  getbits(unsigned x, int p, int n)
{
    return (x>>(p+1-n)) & ~(~0<<n);
}
```

L'espressione x>>(p+1-n) sposta il campo desiderato all'estrema destra della parola. ~0 equivale ad avere tutti i bit pari a 1; spostandolo a sinistra di n posizioni (con l'espressione ~0<<n) si inseriscono degli zeri nei primi n bit di destra; eseguendo poi, sul valore ottenuto, un complemento ad uno (con l'operatore ~), si ottiene una maschera nella quale gli n bit più a destra valgono tutti 1.

Esercizio 2.6 Scrivete una funzione setbits(x, p, n, y) che ritorni la variabile x modificata in modo che i suoi n bit, a partire dalla posizione p, risultino uguali ai primi n bit di destra di y, ed i bit rimanenti restino invariati.

Esercizio 2.7 Scrivete una funzione invert(x, p, n) che ritorni la variabile x modificata in modo che i suoi n bit, a partire dalla posizione p, risultino invertiti (che, cioè, siano stati portati da 0 ad 1 e viceversa), e gli altri bit restino invariati.

Esercizio 2.8 Scrivete una funzione rightrot(x, n) che ritorni il valore dell'intero x ruotato a destra di n posizioni.

2.10 Operatori di Assegnamento ed Espressioni

Espressioni come

```
i=i+2
```

nelle quali la variabile sul lato sinistro viene ripetuta immediatamente sul lato destro, possono essere scritte, usando una forma più compatta, come

```
i+=2
```

L'operatore += è uno degli *operatori di assegnamento*.

Quasi tutti gli operatori binari (che, cioè, hanno un operando sinistro ed uno destro) possiedono un corrispondente operatore di assegnamento, *op*=, dove *op* è uno degli operatori seguenti:

```
+    -    *    /    %    <<    >>    &    ^    |
```

Se *espr_1* ed *espr_2* sono espressioni, allora

espr_1 op= espr_2

equivale a

espr_1 = (espr_1) op (espr-2)

solo che *espr_1*, nel primo caso, viene valutata una volta soltanto. Notate la posizione delle parentesi intorno a *espr_2*:

```
x*=y+1
```

significa

```
x=x*(y+1)
```

e non

```
x=x*y+1
```

Come esempio, consideriamo la funzione `bitcount`, che conta il numero di bit attivi del suo argomento, che dev'essere un intero.

```
/* bitcount: conta i bit attivi di x */
int  bitcount(unsigned x)
{
    int b;

    for (b=0; x!=0; x>>=1)
        if (x&01)
            b++;
    return b;
}
```

Dichiarare l'argomento `x` come `unsigned` assicura che, eseguendo su di esso uno shift a destra, i bit liberatisi vengono azzerati, indipendentemente dal tipo di macchina sulla quale il programma viene eseguito.

Oltre ad essere particolarmente compatti, gli operatori di assegnamento offrono il vantaggio di rispecchiare meglio la struttura del pensiero umano. Noi, infatti, diciamo "aggiungi 2 ad `i`", oppure "incrementa `i` di 2", ma non diciamo mai "prendi `i`, aggiungici 2 e metti il risultato in `i`". Quindi, l'espressione `i+=2` è preferibile all'espressione `i=i+2`. Inoltre, in un'espressione complessa, come

```
yyval[yypv[p3+p4]+yypv[p1+p2]]+=2
```

l'operatore di assegnamento rende più leggibile il codice, poiché il lettore non deve controllare se due espressioni lunghe sono identiche, preoccupandosi eventualmente del fatto che non lo siano. Infine, un operatore di assegnamento può anche aiutare un compilatore a produrre codice più efficiente.

Abbiamo già visto che l'istruzione di assegnamento ha un proprio valore, e può ricorrere all'interno di una espressione; l'esempio più comune di ciò è il seguente:

```
while ((c=getchar())!=EOF)
    ........
```

Anche gli altri operatori di assegnamento (`+=`, `-=`, ecc.) possono ricorrere nelle espressioni, anche se ciò accade più raramente.

In tutte le espressioni di questo genere, il tipo di un'espressione di assegnamento è quello del suo operando sinistro, ed il suo valore è il valore risultante dall'assegnamento stesso.

Esercizio 2.9 In un sistema con numeri in complemento a due, `x&=(x-1)` cancella il bit positivo più a destra in `x`. Spiegate perché. Usate quest'osservazione per scrivere una versione più veloce della funzione `bitcount`.

2.11 Espressioni Condizionali

Le istruzioni

```
if (a>b)
    z=a;
else
    z=b;
```

assegnano a `z`, il massimo fra `a` e `b`. L'*espressione condizionale*, scritta usando l'operatore ternario "`?:`", fornisce un modo alternativo di scrivere questa ed altre costruzioni simili. Nell'espressione

```
espr_1 ? espr_2 : espr_3
```

viene dapprima valutata l'espressione *espr_1*. Se essa ha un valore non nullo (se, cioè, risulta vera), allora viene valutata l'espressione *espr_2*, ed il risultato ottenuto costituisce il valore dell'intera espressione condizionale; in caso contrario, viene valutata *espr_3*, ed il suo valore è anche quello dell'intero costrutto. Soltanto una, fra le espressioni *espr_2* ed *espr_3* viene valutata. Quindi, per calcolare in `z` il massimo fra `a` e `b`, possiamo scrivere:

```
z=(a>b)?a:b;       /* z = max(a,b) */
```

Notate che l'espressione condizionale è essa stessa un'espressione e, in quanto tale, può ricorrere in qualsiasi punto in cui può essere presente un'espressione. Se *espr_2* ed *espr_3* sono di tipo diverso, il tipo del risultato è determinato dalle regole di conversione discusse nei paragrafi precedenti di questo capitolo. Per esempio, se `f` è un `float` ed `n` è un `int`, allora l'espressione

```
(n>0)?f:n
```

è di tipo `float`, indipendentemente dal fatto che `n` sia positivo o meno.

La prima espressione di un'espressione condizionale può non essere racchiusa tra parentesi, perché la precedenza dell'operatore `?:` è molto bassa; in particolare, essa è appena superiore a quella dell'assegnamento. Nonostante ciò, comunque, le parentesi possono essere utili per distinguere meglio la parte condiziona della espressione.

Spesso l'espressione condizionale consente di scrivere codice molto sintetico. Per esempio, questo ciclo stampa `n` elementi di un vettore, 10 per linea, separando ogni colonna con uno spazio bianco, e terminando ogni linea (compresa l'ultima) con un new line.

```
for (i=0; i<n; i++)
    printf("%6d%c", a[i], (i%10==9  ||  i==n-1)?'\n':' ');
```

Dopo ogni gruppo di dieci elementi, viene stampato un new line, che segue anche l'`n`-esimo elemento. Tutti gli altri elementi sono seguiti da uno spazio bianco. Questo codice può sembrare artificioso, ma risulta molto più compatto dell'equivalente `if-else`. Un altro buon esempio è il seguente:

```
printf("Tu hai %d element%s.\n", n, n==1?"o":"i");
```

Esercizio 1.10 Riscrivete la funzione `lower`, che converte lettere maiuscole in lettere minuscole, usando un'espressione condizionale al posto del costrutto `if-else`.

2.12 Precedenza ed Ordine di Valutazione

La Tabella 2.1 riassume le regole di precedenza e di associatività di tutti gli operatori, compresi quelli che non abbiamo ancora discusso. Gli operatori sulla stessa linea hanno la stessa precedenza; le righe sono in ordine di precedenza decrescente, quindi, per esempio, `*`, `/` e `%` hanno la stessa precedenza, che è maggio-re di quella degli operatori binari `+` e `-`. L'"operatore" `()` si riferisce alle chiamate di funzione. Gli operatori `->` e `.` vengono usati per accedere ai membri delle strutture; essi verranno illustrati nel Capitolo 6, insieme a `sizeof` (dimensione di un oggetto). Il Capitolo 5 spiega gli operatori `*`(indirezione tramite un puntatore) e `&` (indirizzo di un oggetto), mentre il Capitolo 3 illustra l'operatore virgola.

Notate che la precedenza degli operatori bit a bit `&`, `^` e `|` è inferiore a quella degli operatori `==` e `!=`. Questo implica che espressioni come

```
if ((x&MASK)==0) ....
```

che controllano lo stato dei bit, debbano essere completamente racchiuse tra parentesi.

Tabella 2.1 *Precedenza e associatività degli operatori*

OPERATORI	ASSOCIATIVITÀ
`() [] -> .`	da sinistra a destra
`! - ++ -- + - * & (tipo) sizeof`	da destra a sinistra
`* / %`	da sinistra a destra
`+ -`	da sinistra a destra
`<< >>`	da sinistra a destra
`< <= > >= == !=`	da sinistra a destra
`&`	da sinistra a destra
`^`	da sinistra a destra
`|`	da sinistra a destra
`&&`	da sinistra a destra
`||`	da sinistra a destra
`?:`	da destra a sinistra
`= += -= *= /= %= &= |= <<= >>=`	da destra a sinistra
`,`	da sinistra a destra

Gli operatori unari `+`, `-` e `*` hanno precedenza maggiore delle rispettive forme binarie.

Il C, come la maggior parte dei linguaggi, non specifica l'ordine in cui vengono valutati gli operandi di un particolare operatore (le eccezioni sono `&&`, `||`, `?:` e `','`). Per esempio, un'istruzione come

```
x=f()+g();
```

`f` potrebbe essere valutata prima di `g`, o viceversa; quindi, se `f` o `g` alterano il valore di qualche variabile dalla quale l'altra funzione dipende, `x` può dipendere dall'ordine di valutazione. Per imporre una particolare sequen-za di valutazione è sufficiente memorizzare in variabili ausiliare i valori intermedi dell'espressione.

Neppure l'ordine di valutazione degli argomenti di una funzione viene specificato, quindi l'istruzione

```
printf("%d %d\n", ++n, power(2,n));        /* SBAGLIATO */
```

può produrre risultati diversi con differenti compilatori, in base al fatto che `n` venga incremento prima o dopo la valutazione di `power`. La soluzione, naturalmente, consiste nello scrivere

```
++n;
printf("%d %d\n", n, power(2,n));
```

Le chiamate a funzioni, le istruzioni di assegnamento nidificate, gli operatori di incremento e di decremento hanno, tutti, degli "effetti collaterali": come risultato della valutazione di un'espressione, qualche variabile viene modificata. In qualsiasi espressione che ha degli effetti collaterali, possono verificarsi sottili dipendenze dall'ordine con il quale le variabili che compaiono nell'espressione vengono aggiornate. Una tipica situazione indesiderata è quella che si verifica nei casi analoghi a quello dell'esempio seguente:

```
a[i]=i++;
```

L'indice del vettore è il vecchio o il nuovo valore di `i`? I compilatori possono interpretare diversamente questa espressione, e produrre di conseguenza dei risultati diversi. Nello standard, molti di questi problemi sono rimasti intenzionalmente imprecisati. Quando si verificano degli effetti collaterali (come l'assegnamento di una variabile), l'interpretazione di ciò che accade viene lasciata al compilatore, perché l'ordine di valutazione migliore può dipendere fortemente dall'architettura della macchina (lo standard specifica che tutti gli effetti

collaterali sugli argomenti devono essere applicati prima che la funzione venga chiamata ma questo, nel caso della `printf` mostrata in precedenza, non è di alcun aiuto).

Come osservazione finale, possiamo dire che scrivere codice dipendente dall'ordine di valutazione è un'abitu-dine scorretta. Naturalmente, è necessario sapere quali azioni evitare ma, se non sapete *come* certe partico-lari azioni vengono attuate sulle diverse macchine, non lasciatevi tentare dal desiderio di avvantaggiarvi della particolare implementazione adottata dal sistema sul quale operate.

CAPITOLO 3

STRUTTURE DI CONTROLLO

In un linguaggio, le istruzioni di controllo del flusso specificano l'ordine secondo il quale i calcoli devono essere effettuati. Negli esempi precedenti abbiamo già visto le più comuni istruzioni per il controllo del flusso; in questo capitolo completeremo la loro panoramica, e saremo più precisi anche in merito ai costrutti già visti.

3.1 Istruzioni e Blocchi

Un'espressione come `x=0`, o `i++` oppure `printf(....)` diventa un'istruzione quando è seguita da un punto e virgola, come in

```
x=0;
i++;
printf(....)
```

In C, il punto e virgola è un terminatore di istruzione e non, come in Pascal, un semplice separatore.

Le parentesi graffe `{` e `}` vengono utilizzate per raggruppare in un'unica *istruzione composta*, detta anche *blocco*, dichiarazioni ed istruzioni, in modo che, dal punto di vista sintattico, esse formino un'entità equivalente ad una sola istruzione. Le parentesi che racchiudono le istruzioni di una funzione sono l'esempio più ovvio di quanto detto; le parentesi che raggruppano alcune istruzioni successive ad un `if`, ad un `else` o ad un `while` costituiscono un altro esempio (le variabili possono essere dichiarate all'interno di un *qualsiasi* blocco, come vedremo nel Capitolo 4). Dopo la parentesi graffa di chiusura di un blocco non esiste punto e virgola.

3.2 IF - ELSE

L'istruzione `if-else` viene usata per esprimere una decisione. Formalmente, la sua sintassi è la seguente:

```
if (espressione)
    istruzione_1
else
    istruzione_2
```

dove la parte associata all'`else` è opzionale. Innanzitutto, l'*espressione* viene valutata; se risulta vera (se, cioè, assume un valore non nullo), viene eseguita l'*istruzione_1*. In caso contrario (se l'*espressione* ha cioè valore nullo) e se esiste la parte `else`, viene eseguita l'*istruzione_2*.

Poiché un `if` non fa altro che controllare il valore numerico di un'espressione, è possibile adottare su di esso delle tecniche di codifica notevolmente compatte. Innanzitutto, è possibile scrivere

```
if (espressione)
```

invece che

```
if (espressione!=0)
```

A volte questo accorgimento risulta chiaro ed intuitivo; in altri casi, però, esso può condurre alla stesura di codice poco leggibile.

Poiché, in un costrutto `if-else`, la parte `else` è opzionale, la mancanza di un `else` all'interno di una sequenza di `if` innestati comporta un'ambiguità, che viene risolta associando ogni `else` all'`if` più interno che ne è privo. Per esempio, in

```
if (n>0)
    if (a>b)
        z=a;
    else
```

```
            z=b;
```

l'`else` viene associato all'`if` più interno, come è evidenziato dall'indentazione. Se volete che ciò non avvenga, dovete scrivere

```
    if (n>0)
    {
        if (a>b)
            z=b;
    }
    else
        z=b;
```

L'ambiguità è particolarmente pericolosa in situazioni analoghe alla seguente:

```
    if (n>=0)
        for (i=0; i<n; i++)
            if (s[i]>0)
            {
                printf("....");
                return i;
            }
    else
        printf("Errore - n è negativo");
```

L'indentazione mostra inequivocabilmente ciò che volete che avvenga, ma il compito non rileva le vostre intenzioni, ed associa l'`else` all'`if` più interno. Questo tipo di errore può essere molto difficile da individuare; di conseguenza, quando si hanno diversi `if` innestati, è bene utilizzare sempre le parentesi.

Per inciso, notate che nell'esempio

```
    if (a>b)
        z=a;
    else
        z=b;
```

`z=a` è seguito da un punto e virgola. Questo accade perché, grammaticalmente, l'`if` è seguito da un'*istruzione*, ed un'istruzione costituita da un'espressione, come "`z=a`", è sempre terminata da un punto e virgola.

3.3 ELSE - IF

La costruzione

```
    if (espressione)
        istruzione
    else if (espressione)
        istruzione
    else if (espressione)
        istruzione
    else if (espressione)
        istruzione
    else
        istruzione
```

ricorre tanto frequentemente da richiedere una discussione particolareggiata. Questa sequenza di istruzioni di `if` è il modo più generale di realizzare una scelta plurima. Le *espressioni* vengono valutate nell'ordine in cui si presentano; se una di esse risulta vera, l'*istruzione* ad essa associata viene eseguita, e ciò termina la intera catena. Come sempre, il codice corrispondente ad ogni *istruzione* può essere un'istruzione singola o un gruppo di istruzioni racchiuse tra parentesi graffe.
L'ultimo `else` gestisce la condizione "nessuna delle precedenti", cioè il caso di default, eseguito quando nessuna delle espressioni risulta vera. Può accadere che al caso di default non corrisponda alcuna azione esplicita; in questa situazione, il blocco finale

```
    else
```

```
        istruzione
```

può essere tralasciato, oppure può venire utilizzato per controllare condizioni di errore.

Per illustrare un caso di decisione a tre vie, presentiamo una funzione di ricerca binaria. Tale funzione determina se un particolare valore `x` è presente nel vettore ordinato `v`. Gli elementi di `v` devono essere disposti in ordine crescente. La funzione restituisce la posizione (un numero compreso tra `0` e `n-1`) di `x` compare in `v`, -1 altrimenti.

L'algoritmo di ricerca binaria confronta dapprima `x` con l'elemento centrale di `v`. Se `x` è minore del valore contenuto in tale elemento, la ricerca prosegue sulla metà inferiore del vettore; se `x` è, invece, maggiore dello elemento centrale, la ricerca continua sulla parte superiore di `v`. Questo processo di bisezione prosegue fino al reperimento di `x` oppure fino all'esaurimento del vettore.

```
/* binsearch: trova x in v[0] <= v[1] <= .... <= v[n-1] */
int  binsearch(int x, int v[], int n)
{
    int low, high, mid;

    low=0;
    high=n-1;
    while (low<=high)
    {
        mid=(low+high)/2;
        if (x<v[mid])
            high=mid-1;
        else if (x>v[mid])
            low=mid+1;
        else
            return mid;       /* trovato */
    }
    return -1;                /* non trovato */
}
```

Il punto focale consiste nel decidere se `x` è minore, maggiore o uguale dell'elemento centrale di ogni iterazione, `v[mid]`; questo è un tipico problema risolvibile con il costrutto `if-else`.

Esercizio 3.1 Il nostro algoritmo di ricerca binaria effettua due test all'interno di ogni ciclo di `while`, mentre un test unico sarebbe sufficiente (pur di aumentare i test all'esterno del `while`). Scrivete una versione con un test soltanto all'interno del `while`, e misurare la differenza nel tempo di esecuzione delle due funzioni.

3.4 SWITCH

L'istruzione `switch` è una struttura di scelta plurima che controlla se un'espressione assume un valore allo interno di un certo insieme di *costanti* intere, e si comporta di conseguenza.

```
switch (espressione)
{
    case espr-cost : istruzioni
    case espr-cost : istruzioni
    default : istruzioni
}
```

Ogni caso possibile è etichettato da un insieme di costanti intere e di espressioni costanti. Se il valore di *espressione* coincide con uno di quelli contemplati nei vari casi, l'esecuzione inizia da quel caso particolare. Le espressioni contemplate nei diversi casi devono essere differenti. L'ultimo caso, identificato dall'etichetta `default`, viene eseguito solo se nessuno dei casi precedenti è stato soddisfatto, ed è opzionale. Se esso non compare, e nessuno dei casi elencati si verifica, non viene intrapresa alcuna particolare azione. Le clausole `case` e `default` possono occorrere in un ordine qualsiasi.

Nel Capitolo 1 abbiamo scritto un programma per contare le occorrenze di ogni cifra, degli spazi e di tutti gli altri caratteri, usando una sequenza del tipo `if .... else if .... else`. Riscriviamolo, ora, utilizzando il costrutto `switch`:

```
#include  <stdio.h>

main()      /* conta cifre, spazi e tutti gli altri caratteri */
{
    int c, i, nwhite, nother, ndigit[10];

    nwhite=nother=0;
    for (i=0; i<10; i++)
        ndigit[i]=0;
    while ((c=getchar())!=EOF)
    {
        switch(c)
        {
            case '0': case '1': case '2': case '3': case '4':
            case '5': case '6': case '7': case '8': case '9':
                ndigit[c-'0']++;
                break;
            case ' ':
            case '\n':
            case '\t':
                nwhite++;
                break;
            default:
                nother++;
                break;
        }
    }
    printf("cifre =");
    for (i=0; i<10; i++)
        printf(" %d", ndigit[i]);
    printf(", spazi = %d, altri = %d\n", nwhite, nother);
    return 0;
}
```

L'istruzione `break` provoca l'uscita (exit) immediata dallo `switch`. Poiché i casi servono soltanto come etichette, l'esecuzione delle istruzioni associate ad uno di essi è seguita dall'*esecuzione sequenziale* dei casi successivi, a meno che non sia presente un'istruzione esplicita di uscita. `break` e `return` sono le due forme più usate per l'uscita dallo `switch`. Come vedremo nel prossimo capitolo, un'istruzione `break` può essere impiegata anche per uscire prematuramente da un `while`, da un `for` e da un `do`.

Il fatto che, a meno di direttive esplicite, l'esecuzione prosegua fino al termine dello `switch`, comporta vantaggi e svantaggi. Da un lato, ciò consente di associare più possibilità ad un'unica etichetta, come nel caso delle cifre dell'esempio precedente. Dall'altro, una simile logica costringe a specificare il `break` ogni volta che si vuole impedire l'esecuzione di più casi. Inoltre, il fatto di eseguire in cascata più di un caso può comportare notevoli problemi in caso di modifica del programma originale. Ad eccezione dei casi nei quali si vogliono avere più etichette associate ad un'unica azione, l'esecuzione in cascata dovrebbe essere evitata e, ove risultasse necessaria, dovrebbero essere sempre posti dei commenti.

Anche se è inutile dal punto di vista logico, è infine bene inserire sempre un `break` al termine delle istruzioni associate all'etichetta `default`. Nel caso in cui, in seguito, venissero inseriti casi dopo il `default`, questa precauzione preserverebbe la correttezza del vostro programma.

Esercizio 3.2 Scrivete una funzione `escape (s, t)` che converte i caratteri come new line e il carattere di tabulazione in sequenze di escape visibili, come `\n` e `\t`, mano a mano che li incontra durante la copia della stringa `t` nella stringa `s`. Usate uno `switch`. Scrivete una funzione opposta, che converta le sequenze di escape in caratteri reali.

3.5 Cicli - WHILE e FOR

Nel corso di questo libro, abbiamo già incontrato i cicli `while` e `for`. Nel caso di

```
while (espressione)
    istruzione
```

espressione viene valutata. Se il suo valore è diverso da zero, viene eseguita l'*istruzione* specificata e la *espressione* viene valutata nuovamente. Il ciclo continua fino a quando l'*espressione* risulta falsa, ed allora la esecuzione riprende dalla prima istruzione non controllata dal `while`.

Il costrutto `for`

```
for (espr_1; espr_2; espr_3)
    istruzione
```

equivale a

```
espr_1;
while (espr_2)
{
    istruzione
    espr_3;
}
```

fatta eccezione per l'esecuzione dell'istruzione `continue`, che verrà descritta nella Sezione 3.7.

Grammaticalmente, le tre componenti di un ciclo `for` sono delle espressioni. In genere, *espr_1* ed *espr_3* sono degli assegnamenti o delle chiamate di funzione, mentre *espr_2* è un'espressione relazionale. Ognuna delle tre parti può venire tralasciata, anche se i punti e virgola devono sempre essere presenti. L'assenza di *espr_1* o di *espr_3* si traduce, semplicemente, in una mancata espansione del codice al momento della compilazione, Se, invece, la parte mancante è il test, cioè *espr_2*, il compilatore assume che esso sia sempre vero, quindi

```
for (;;)
{
    ....
}
```

è un ciclo infinito che, presumibilmente, verrà interrotto con l'impiego di istruzioni particolari come `break` o `return`.

Il fatto di usare un `while` piuttosto che un `for`, dipende largamente dalle preferenze personali. Per esempio, in

```
while ((c=getchar()==' '  ||  c=='\n'  ||  c=='\t')
    ;        /* tralascia i caratteri di spaziatura */
```

non esistono fasi di inizializzazione o di re-inizializzazione, perciò il costrutto più adatto è, senza ombra di dubbio, il `while`.

L'impiego del `for` è preferibile quando sono presenti inizializzazioni semplici ed incrementi, perché questo costrutto raggruppa le istruzioni di controllo e, ponendole all'inizio del ciclo, le evidenza molto bene. L'uso più frequente del `for`, infatti, è il seguente:

```
for (i=0; i<n; i++)
    ....
```

che è la classica costruzione C usata per elaborare i primi `n` elementi di un vettore e che corrisponde al ciclo `DO` del Fortran ed al `for` del Pascal. Quest'ultima analogia, però, non è perfetta, in quanto l'indice e la condizione limite di un ciclo di `for`, in C, possono essere modificati all'interno del ciclo stesso, e la variabile `i`, usata come indice, conserva il proprio valore anche dopo la terminazione del ciclo. Poiché le componenti del `for` sono espressioni arbitrarie, i cicli di `for` non sono applicabili soltanto a progressioni aritmetiche. In ogni caso, è meglio evitare di inserire dei calcoli non strettamente correlati fra loro nelle parti di inizializzazione e di incremento di un `for`, poiché esse sono logicamente riservate alle operazioni di controllo.

A titolo di esempio, presentiamo una nuova versione della funzione `atoi`, che converte una stringa di cifre nel suo equivalente valore numerico. Questa versione è leggermente più generale di quella presentata nel

Capitolo 2; essa gestisce eventuali spazi bianchi ed il segno `+` o `-`, opzionale (nel Capitolo 4 illustreremo la funzione `atof`, che converte la stringa nel suo valore in floating-point).

La struttura del programma riflette il formato dell'input

```
tralascia, se esistono, gli spazi bianchi
se c'è, preleva il segno
preleva la parte intera e convertila
```

Ogni fase svolge dei compiti specifici, e termina lasciando una situazione corretta alla quale applicare la fase successiva. L'intero processo non appena viene individuato un carattere che non può appartenere ad un numero.

```
#include  <ctype.h>

/* atoi: converte s in un intero; versione 2 */
int  atoi(char s[])
{
    int i, n, sign;

    for (i=0; isspace(s[i]); i++)   /* tralascia gli spazi */
        ;
    sign=(s[i]=='-')?-1:1;
    if (s[i]=='+'  ||  s[i]=='-')   /* tralascia il segno */
        i++;
    for (n=0; isdigit(s[i]); i++)
        n=10*n+(s[i]-'0');
    return sign*n;
}
```

Per la conversione di stringhe in interi, la libreria standard fornisce una funzione più complessa, `strtol`; a questo proposito, si veda la Sezione 5 dell'Appendice B.

I vantaggi derivanti dal raggruppamento delle istruzioni di controllo risultano ancor più evidenti in presenza di cicli nidificati. La funzione che segue è un algoritmo di Shell sort per l'ordinamento di un vettore di interi. La idea di base di questo algoritmo, inventato da D. L. Shell nel 1959, consiste nel confrontare innanzitutto elementi del vettore molto distanti fra loro, contrariamente a quanto avviene negli algoritmi di ordinamento tradizionali, dove i confronti avvengono tra elementi adiacenti. L'approccio di Shell consente di eliminare velocemente la maggior parte delle condizioni di disordine, in modo che, all'aumentare delle iterazioni, il lavoro da svolgere diminuisca. La distanza fra gli elementi messi a confronto viene gradualmente decrementata di una unità, fino a che le operazioni di ordinamento non diventano, di fatto, degli scambi fra elementi adiacenti.

```
/* shellsort: ordina v[0] .... v[n-1] in ordine crescente */
void  shellsort(int v[], int n)
{
    int gap, i, j, temp;

    for (gap=n/2; gap>0; gap/=2)
        for (i=gap; i<n; i++)
            for (j=i-gap; j>=0  &&  v[j]>v[j+gap]; j-=gap)
            {
                temp=v[j];
                v[j]=v[j+gap];
                v[j+gap]=temp;
            }
}
```

In questa funzione troviamo tre cicli innestati. Il ciclo più esterno controlla la distanza tra i due elementi di volta in volta a confronto, inizializzandola a `n/2` e decrementandola di un fattore due fino a che essa non diventa nulla. Il ciclo intermedio scandisce tutti gli elementi. Il ciclo più interno, infine, confronta ogni coppia di elementi separati da `gap` posizioni e, se tali elementi sono disordinati, li inverte. Poiché, dopo un certo numero di iterazioni, `gap` varrà sicuramente uno, tutti gli elementi, al termine dell'algoritmo, risulteranno ordinati correttamente. Notate come la generalità del `for` consenta al ciclo più esterno, che non è una progressione aritmetica, di adattarsi comunque alla sintassi di questo costrutto.

Il C prevede anche l'operatore virgola ",", che spesso viene utilizzato nell'istruzione `for`. Una coppia di espressioni separate da una virgola viene valutata da sinistra a destra, ed il tipo ed il valore del risultato coincidono con il tipo ed il valore dell'espressione di destra. Quindi, in ogni sezione della parte di controllo di un `for` è possibile inserire delle espressioni multiple, per esempio per incrementare parallelamente due indici. Questo aspetto è illustrato nell'esempio seguente, che mostra una funzione `reverse(s)` che inverte la stringa `s`.

```
#include  <string.h>

/* reverse: inverte la stringa s */
void  reverse(char s[])
{
    int c, i, j;

    for (i=0, j=strlen(s)-1; i<j; i++, j--)
    {
        c=s[i];
        s[i]=s[j];
        s[j]=c;
    }
}
```

Le virgole che separano gli argomenti di una funzione o le variabili di una dichiarazione *non* rappresentano lo operatore virgola, e non garantiscono la valutazione da sinistra a destra.

L'operatore virgola dovrebbe essere usato con molta cautela. L'uso migliore che se ne può fare è nei costrutti strettamente correlati gli uni agli altri, come nel caso del `for` della funzione `reverse`, e nelle macro, dove un'elaborazione in più fasi deve apparire sotto forma di espressione singola. Un'espressione con l'operatore virgola potrebbe rivelarsi appropriata anche nel caso dello scambio di due elementi, che può essere pensato come una singola operazione:

```
for (i=0, j=strlen(s)-1; i<j; i++, j--)
    c=s[i], s[i]=s[j], s[j]=c;
```

Esercizio 3.3 Scrivete una funzione `expand(s1, s2)` che espande le notazioni abbreviate presenti nella stringa `s1` in notazioni estese (per esempio, traduce `a-z` in `abc...xyz`) e pone in `s2` la stringa risultante. `s1` può essere costituita da caratteri maiuscoli, minuscoli e da cifre; gestite abbreviazioni quali `a-b-c`, `a-z0-9` e `-a-z`. Assicuratevi che un `-` iniziale o finale venga interpretato letteralmente.

3.6 Cicli - DO - WHILE

Come abbiamo visto nel Capitolo 1, i costrutti `while` e `for` controllano la condizione di terminazione all'inizio del ciclo. Al contrario, il terzo tipo di ciclo del C, il `do-while`, controlla la condizione di uscita *al termine* di ogni iterazione; quindi, il corpo del ciclo viene sempre eseguito almeno una volta.

La sintassi del `do` è la seguente:

```
do
    istruzione
while (espressione);
```

Dapprima viene eseguita l'*istruzione*, e soltanto successivamente l'*espressione* viene valutata. Se essa risulta vera, l'*istruzione* viene eseguita nuovamente. Quando l'*espressione* diventa falsa, il ciclo termina. A parte il senso del test, il `do-while` del C è equivalente al `repeat-until` del Pascal.

L'esperienza dimostra che il `do-while` è meno usato del `while` e del `for`. Nonostante ciò, è bene valutare di volta in volta quale sia il costrutto più adatto alla situazione; per esempio, nella funzione `itoa`, che converte un numero in una stringa di caratteri (l'inverso della funzione `atoi`), il `do-while` si rivela la struttura di controllo più idonea alla soluzione del problema. Il compito della funzione `itoa` è più complesso di quanto po-trebbe apparire inizialmente, perché gli algoritmi più semplici per la generazione delle cifre le generano in ordine inverso. Noi abbiamo scelto dapprima la stringa invertita e, in seguito, convertita.

```
/* itoa: converte n in una stringa s */
void  itoa(int n, char s[])
{
    int i, sign;

    if ((sign=n)<0)           /* memorizza il segno */
        n=-n;                 /* trasforma n in un numero positivo */
    i=0;
    do                        /* genera le cifre in ordine inverso */
    {
        s[i++]=n%10+'0';      /* legge la cifra successiva */
    } while ((n/=10)>0);      /* la cancella */
    if (sign<0)
        s[i++]='-';
    s[i]='\n';
    reverse(s);
}
```

Il `do-while` è necessario o, almeno, conveniente, perché nel vettore `s` deve sempre essere inserito almeno un carattere, anche se `n` è zero. Da un punto di vista sintattico, le parentesi graffe che abbiamo usato per racchiudere il corpo del `do-while` sono superflue, ma evidenziano il fatto che l'istruzione di `while` rappresenta la fine di un `do`, anziché l'inizio di un `while`.

Esercizio 3.4 In una rappresentazione in complemento a due, la versione di `itoa` non gestisce il numero negativo con modulo massimo, dato da `n=-(2^(wordsize-1))`, dove `wordsize` è il numero di bit che compongono la word. Spiegate il motivo di questa carenza. Modificate la funzione in modo che stampi correttamente tutti i valori, indipendentemente dall'architettura della macchina sulla quale viene eseguita.

Esercizio 3.5 Scrivete una funzione `itob(n, s, b)` che converte l'intero `n` nella corrispondente sequenza di caratteri in base `b`, e mette in `s` la stringa risultante. In particolare, `itob(n, s, 16)` trasforma `n` in un intero esadecimale e lo scrive in `s`.

Esercizio 3.6 Scrivete una versione di `itoa` che accetti tre argomenti invece di due. Il terzo argomento è il numero minimo di caratteri di cui dev'essere composta la stringa finale; quindi il numero convertito, se necessario, deve essere preceduto dall'opportuna quantità di spazi bianchi.

3.7 BREAK e CONTINUE

Talvolta può essere utile uscire da un ciclo senza controllare, all'inizio o alla fine dell'iterazione, la condizione di terminazione. L'istruzione `break` provoca l'uscita incondizionata da un `for`, un `while` oppure un `do`, nello stesso modo in cui consente di uscire da uno `switch`. Un `break`, infatti, provoca l'uscita dallo `switch` oppure dal ciclo più interno che lo contiene.

La funzione che segue, `trim`, rimuove dalla fine di una stringa gli spazi, i caratteri di tabulazione ed i new line; per farlo, essa usa un `break` che le consente di uscire dal ciclo non appena, scandendo la stringa da destra a sinistra, trova un carattere diverso dallo spazio, dal tab e dal new line.

```
/* trim: rimuove gli spazi, i tab ed i new line */
int  trim(char s[])
{
    int n;

    for (n=strlen(s)-1; n>=0; n--)
        if (s[n]!=' '  &&  s[n]!='\t'  &&  s[n]!='\n')
            break;
    s[n+1]='\0';
    return n;
}
```

`strlen` restituisce la lunghezza della stringa. Il ciclo di `for` inizia al termine della stringa, e la scandisce a ritroso fino a trovare un carattere diverso dallo spazio, dal tab e dal new line. Il ciclo viene interrotto con un `break` quando un simile carattere viene trovato, oppure quando `n` diventa negativo (cioè quando l'intera

stringa è stata scandita). Verificate che il comportamento della funzione è corretto anche nei casi limite, nei quali la stringa è composta da soli spazi oppure è vuota.

L'istruzione `continue` è legata al `break`, ma viene usata più raramente; essa forza l'inizio dell'iterazione successiva di un `for`, un `while` oppure un `do`; essa, cioè, provoca l'esecuzione immediata della parte di controllo del ciclo. Nel caso del `for`, il controllo passa alla parte di incremento. A differenza del `break`, la istruzione `continue` non è applicabile allo `switch`. Un `continue` all'interno di uno `switch` inserito in un ciclo provoca il passaggio alla successiva iterazione del ciclo stesso.

A titolo di esempio, mostriamo il seguente frammento di codice, che tratta soltanto gli elementi positivi o nulli del vettore `s`, tralasciando quelli negativi.

```
for (i=0; i<n; i++)
{
    if (a[i]<0)          /* tralascia gli elementi negativi */
        continue;
    ........             /* tratta gli elementi non negativi */
}
```

L'istruzione `continue` viene spesso utilizzata quando la parte successiva del ciclo è particolarmente complicata; in questo caso, infatti, invertire un test e creare un altro livello di indentazione potrebbe condurre alla stesura di codice troppo nidificato.

3.8 GOTO e LABEL

Il C fornisce l'abusata istruzione `goto`, e le label ("etichette") per potere ramificare l'esecuzione. Formalmente, il `goto` non è mai necessario, e nella pratica spesso la stesura di codice che non lo utilizza risulta meno difficoltosa. In questo libro non abbiamo mai usato il `goto`.

Nonostante ciò, ci sono alcune rare situazioni nelle quali l'impiego del `goto` può rivelarsi conveniente. Il caso più comune si verifica quando è necessario uscire contemporaneamente da due o più cicli innestati. L'istruzione `break`, uscendo soltanto dal ciclo più interno, si rivela in questo caso insufficiente. Quindi:

```
for ( .... )
    for ( .... )
    {
        ....
        if (disaster)
            goto error;
    }
....

error:
    ripristina la situazione
```

Quest'organizzazione è utile quando la gestione delle condizioni di errore non è banale, e quando gli errori si possono verificare in più punti del codice.

Una label ha la stessa forma sintattica di una variabile, ed è seguita da un due punti. Essa può essere associata a qualsiasi istruzione della funzione contenente il `goto`. Il suo scope è l'intera funzione.

Come ulteriore esempio, consideriamo il problema di determinare se due vettori, `a` e `b`, hanno un elemento in comune. Una possibile soluzione è:

```
    for (i=0; i<n; i++)
        for (j=0; i<m; j++)
            if (a[i]==b[j])
                goto found;
    /* non sono stati trovati elementi comuni */
    ....
found:
    /* trovato un elemento comune: a[i]==b[j] */
    ....
```

Qualsiasi segmento di codice contenente un `goto` può sempre essere scritto anche senza di esso, anche se a volte, per farlo, è necessario ripetere dei controlli o introdurre un'ulteriore variabile. Per esempio, il codice precedente può diventare:

```
found=0;
for (i=0; i<n  &&  !found; i++)
    for (j=0; i<m  &&  !found; j++)
        if (a[i]==b[j])
            found=1;
if (found)
    /* trovato un elemento: a[i-1]==b[j-1] */
    ....
else
    /* non sono stati trovati elementi comuni */
    ....
```

Ad esclusione dei rari casi citati in precedenza, il codice che comprende dei `goto` è meno comprensibile e più difficilmente manutenibile di quello che ne è privo. Pur non volendo essere dogmatici a questo proposito, consigliamo di usare i `goto` con molta attenzione e, se possibile, evitarli del tutto.

CAPITOLO 4

FUNZIONI E STRUTTURA DEI PROGRAMMI

Le funzioni consentono di scomporre problemi complessi in moduli più semplici, sfruttabili anche singolarmente per la risoluzione di problemi diversi. Se strutturate nel modo corretto, le funzioni rendono invisibili al resto del programma i dettagli implementativi che non è necessario esso conosca; in tal modo, esse rendono più chiaro il programma nel suo complesso e ne facilitano notevolmente la manutenzione.

Il C è stato ideato con l'intenzione di rendere le funzioni efficienti e facilmente utilizzabili; più che da poche funzioni molto complesse, i programmi C sono in genere costituiti da molte funzioni piuttosto semplici. Un programma può risiedere in uno o più file sorgente, che possono essere compilati separatamente e caricati insieme unitamente, anche, a funzioni di libreria compilate in precedenza. Noi non approfondiremo questo argomento, poiché esso è fortemente legato all'architettura del sistema sul quale si opera.

La dichiarazione e la definizione di funzione sono le aree nelle quali lo standard ANSI ha apportato le modifiche più significative al C. Come abbiamo visto nel Capitolo 1, ora è possibile dichiarare il tipo degli argomenti di una funzione al momento della sua dichiarazione. Anche la sintassi della definizione di funzione cambia, in modo da restare coerente con la dichiarazione. Questa nuova sintassi consente ad un compilatore di rilevare molti errori che, con la vecchia versione del C, non erano evidenziabili. Inoltre, quando gli argomenti sono dichiarati correttamente, le conversioni di tipo vengono effettuate in modo automatico.

Lo standard precisa le regole sullo scope dei nomi; in particolare, esso richiede che ogni oggetto dichiarato esternamente abbia una propria definizione. L'inizializzazione è più generale: ora i vettori e le strutture automatiche possono essere inizializzati esplicitamente.

Anche il preprocessore C è stato migliorato. Ora le sue funzionalità comprendono un insieme completo di direttive per la compilazione condizionale, uno strumento per creare costanti di tipo stringa (quindi tra doppi apici) mediante l'utilizzo delle macro, ed un migliore controllo sul processo di espansione delle macro stesse.

4.1 Fondamenti sulle Funzioni

Per iniziare, progettiamo e scriviamo un programma che stampa tutte le linee del suo input contenenti una stringa particolare (questo è un caso particolare del programma UNIX `grep`). Per esempio, se la stringa voluta è "`ould`", il seguente input:

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

produrrà l'output

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

Il problema è nettamente separato in tre fasi:

```
while (c'è un'altra linea)
    if (la linea contiene la stringa voluta)
        stampala
```

Nonostante sia certamente possibile inserire tutto il codice richiesto all'interno della funzione `main`, è meglio sfruttare questa netta separazione delle operazioni, scrivendo una funzione apposita per ogni fase. Infatti, tre brevi segmenti di codice sono più maneggevoli di un singolo elemento molto esteso, perché i dettagli non significativi possono essere "nascosti" nelle diverse funzioni, minimizzando così la possibilità di interazioni non desiderate. Infine, i singoli segmenti possono facilmente essere riutilizzati in altri programmi.

La frase “fino a quando c’è un’altra linea” si traduce nella funzione `getline`, che abbiamo scritto nel Capitolo 1, mentre la frase “stampala” non è altro che la `printf`, scritta per noi da qualcun altro. Questo significa che l’unica parte mancante è la funzione che decide se la linea di input contiene la stringa cercata.

Possiamo risolvere il problema scrivendo la funzione `strindex (s, t)`, che restituisce la posizione, cioè lo indice, di inizio della stringa `t` all’interno del vettore `s`, oppure `-1` se `t` non compare in `s`. Poiché, in C, gli indici di un vettore partono da zero, il fatto che `strindex` ritorni un valore negativo indica inequivocabilmente che la linea di input non contiene la stringa che cerchiamo. Quando, in seguito, avremo bisogno di risolvere problemi più complessi di ricerca su stringhe, non dovremo fare altro che sostituire la funzione `strindex`, lasciando invariato il resto del codice (la libreria standard fornisce la funzione `strstr`, molto simile a `strindex`, ma che restituisce un puntatore alla posizione del vettore).

Una volta definita questa struttura generale, l’implementazione del programma risulta più semplice. Di seguito presentiamo il programma completo, in modo che possiate osservare la complementarietà dei diversi seg-menti. Per ora, imponiamo che la stringa da ricercare sia una costante letterale, che non è il meccanismo più generale adottabile. In seguito, torneremo brevemente sul problema dell’inizializzazione dei vettori di ca-ratteri, e nel Capitolo 5 mostreremo come trasformare la stringa in un parametro, inizializzato al momento dell’esecuzione del programma. Notate, infine, che questa versione di `getline` è leggermente diversa da quella presentata nel Capitolo 1; il confronto tra le due versioni potrebbe risultare, per voi, molto istruttivo.

```
#include  <stdio.h>
#define  MAXLINE  1000  /* massima lunghezza di una linea di input */

int  getline(char line[], int max);
int  strindex(char source[], char searchfor[]);

char  pattern[]=”ould”;       /* stringa da ricercare */

/* trova tutte le linee contenenti la stringa voluta */
main()
{
    char line[MAXLINE];
    int found=0;

    while (getline(line, MAXLINE)>0)
        if (strindex(line, pattern)>=0)
        {
            printf(“%s”, line);
            found++;
        }
    return found;
}

/* getline: carica una linea in s, ritorna la lunghezza */
int  getline(char s[], int lim)
{
    int c, i;

    i=0;
    while (--lim>0  &&  (c=getchar())!=EOF  &&  c!=’\n’)
        s[i++]=c;
    if (c==’\n’)
        s[i++]=c;
    s[i]=’\0’;
    return i;
}

/* strindex: ritorna l’indice di t in s, -1 se t non compare */
int  strindex(char s[], char t[])
{
    int i, i, k;
    for (i=0; s[i]!=’\0’; i++)
    {
        for (j=i, k=0; t[k]!=’\0’  &&  s[j]==t[k]; j++, k++)
```

```
            ;
        if (k>0  &&  t[k]=='\0')
            return i;
    }
    return -1;
}
```

Ogni definizione di funzione ha la forma:

```
tipo-ritornato  nome-funzione(dichiarazioni argomenti)
{
    dichiarazioni ed istruzioni
}
```

Alcune di queste parti possono non comparire; la funzione minimale è, infatti, la seguente

```
dummy() { }
```

che non fa nulla e non ritorna alcun valore. Durante lo sviluppo di un programma, una funzione come questa può servire come sostituto temporaneo di funzioni che verranno scritte in un secondo tempo. Se viene omesso il tipo del valore di ritorno, esso viene considerato automaticamente un `int`.

Un programma non è altro che un insieme di definizioni di variabili e di funzioni. La comunicazione tra le funzioni avviene tramite gli argomenti, i valori di ritorno delle funzioni stesse e le variabili esterne. All'interno del file sorgente, le funzioni possono essere disposte in un ordine qualsiasi, ed il programma sorgente può essere suddiviso in diversi file, cosa che non può avvenire per una singola funzione, che deve trovarsi in un unico file.

L'istruzione `return` è il meccanismo che consente alla funzione chiamata di restituire un valore al chiamante. Quest'istruzione può essere seguita da una qualsiasi espressione:

```
return espressione;
```

Se necessario, l'*espressione* verrà convertita nel tipo del valore ritornato dalla funzione. Per motivi di chiarezza, l'*espressione* viene spesso racchiusa fra parentesi tonde, che però sono opzionali.

La funzione chiamante è libera di trascurare il valore restituito. In tal caso, il `return` all'interno della funzione chiamata non è seguito da alcuna espressione, ed il chiamante non riceve alcun valore di ritorno. Un altro caso in cui il chiamante riprende il controllo dell'esecuzione, senza che gli venga passato alcun valore di ritorno, è quello in cui l'esecuzione della funzione chiamata termina per il raggiungimento della parentesi graffa di chiusura. Questa situazione non è illegale ma, in genere, il fatto che una funzione, in un punto, ritorni un valore ed in un altro non ritorni nulla, è indice di una situazione anomala. In ogni caso se, per un qualunque motivo, una funzione non riesce a restituire il proprio valore, ciò che il chiamante riceve è un valore "sporco".

Il `main` del programma di ricerca del pattern restituisce il numero di occorrenze trovate. Questo valore viene perciò messo a disposizione dell'ambiente che ha eseguito il programma.

Il metodo per compilare e caricare un programma C distribuito su più file sorgente varia da un sistema all'altro. Sul sistema UNIX, per esempio, questo compito viene assolto dal comando `cc`, citato nel Capitolo 1. Supponiamo che le tre funzioni del programma di ricerca del pattern siano scritte in tre diversi file sorgente, rispettivamente `main.c`, `getline.c` e `strindex.c`. Allora il comando

```
cc  main.c  getline.c  strindex.c
```

compila i tre file, ponendo dapprima i risultati di ogni compilazione nei tre file `main.o`, `getline.o` e `strindex.o`; quindi `cc` carica questi tre file in un unico file eseguibile, che chiama `a.out`. Se in un file, per esempio in `main.c`, c'è un errore, tale file può essere ricompilato separatamente e caricato insieme ai file oggetto generati in precedenza, con il comando

```
cc  main.c  getline.o  strindex.o
```

Il comando `cc` sostituisce al suffisso "`.c`" per distinguere un file sorgente dal suo oggetto.

Esercizio 4.1 Scrivete la funzione `strrindex(s, t)`, che restituisce la posizione dell'occorrenza *più a destra* di `t`, oppure `-1` se `t` non compare in `s`.

4.2 Funzioni Che Ritornano Valori Non Interi

Fino a questo momento, le funzioni dei nostri esempi sono sempre state di due tipi: o non restituivano alcun valore (`void`), oppure restituivano un intero (`int`). Cosa accade se una funzione deve ritornare un valore di qualche altro tipo? Molte funzioni numeriche, come `sqrt`, `sin` e `cos`, ritornano un `double`; altre funzioni specializzate restituiscono valori di tipo diverso. Per illustrare il trattamento di queste funzioni, scriviamo ed usiamo la funzione `atof(s)`, che converte la stringa `s` nel corrispondente numero floating-point in doppia precisione. `atof` è un'estensione della funzione `atoi`, che abbiamo presentato in versioni diverse nei Capitoli 2 e 3. `atof` gestisce un segno (opzionale) ed il punto decimale, oltre che la presenza o l'assenza della parte frazionaria o di quella intera. La nostra non è una routine di conversione dell'input di qualità molto elevata; scrivere una funzione di ottimo livello, infatti, richiederebbe un impiego eccessivo di spazio e di tempo. In ogni caso una funzione `atof` è inclusa nella libreria standard e dichiarata nell'header `<math.h>`.

Come prima cosa, `atof`, che non restituisce un `int`, deve dichiarare il tipo del suo valore di ritorno. Il nome del tipo precede il nome della funzione:

```
#include  <ctype.h>

/* atof: converte la stringa s in un double */
double  atof(char s[])
{
    double val, power;
    int i, sign;

    for (i=0; isspace(s[i]); i++)   /* tralascia gli spazi */
        ;
    sign=(s[i]=='-')?-1:1;
    if (s[i]=='+'  ||  s[i]=='-')
        i++;
    for (val=0.0; isdigit(s[i]); i++)
        val=10.0*val+(s[i]-'0');
    if (s[i]=='.')
        i++;
    for (power=1.0; isdigit(s[i]); i++)
    {
        val=10.0*val+(s[i]-'0');
        power*=10.0;
    }
    return sign*val/power;
}
```

In secondo luogo, la funzione chiamante deve sapere che `atof` restituisce un valore che non è un `int`. Un modo per assicurarsi che ciò avvenga consiste nel dichiarare esplicitamente `atof` all'interno del chiamante. Tale dichiarazione è mostrata in questo rudimentale programma di calcolo, che per ogni linea di input legge un numero, che può essere preceduto da un segno, e lo somma ai numeri letti in precedenza, stampando il risultato dopo ogni linea letta.

```
#include  <stdio.h>
#define  MAXLINE  100

/* rudimentale calcolatrice */
main()
{
    double sum, atof(char[]);
    char line[MAXLINE];
    int getline(char line[], int max);

    sum=0;
    while (getline(line, MAXLINE)>0)
        printf("\t%g\n", sum+=atof(line));
    return 0;
```

```
}
```

La dichiarazione

```
double sum, atof(char[]);
```

afferma che `sum` è una variabile di tipo `double`, e che `atof` è una funzione che riceve in input un argomento di tipo `char[]` e restituisce un `double`.

La dichiarazione e la definizione della funzione `atof` devono essere consistenti. Se la funzione `atof` e la sua chiamata, in `main`, avessero due tipi inconsistenti fra loro e si trovassero in un unico file sorgente, il compilatore rileverebbe l'errore. Ma se (come in genere accade) `atof` fosse stata compilata separatamente, l'inconsistenza non verrebbe rilevata, `atof` restituirebbe un `double` che il `main` tratterebbe come un `int`, ed il risultato finale sarebbe privo di significato.

Alla luce di quanto abbiamo fino ad ora detto sulla coerenza richiesta fra dichiarazioni e definizioni, un simile comportamento può sorprendere. La ragione per la quale si può verificare un'inconsistenza risiede nel fatto che, in mancanza di un prototipo della funzione, quest'ultima viene dichiarata implicitamente dalla sua prima occorrenza in un'espressione, come in

```
sum+=atof(line);
```

Se, in un'espressione, ricorre un nome che non è stato dichiarato in precedenza e tale nome è seguito da una parentesi tonda sinistra, esso, dal contesto, viene dichiarato automaticamente come nome di una funzione, che si assume ritorni un valore intero; per quanto concerne gli argomenti della funzione, invece, non viene fatta alcuna assunzione. Anche una dichiarazione di funzione che non contenga gli argomenti, come in

```
double  atof();
```

indica esplicitamente che sugli argomenti stessi non dev'essere fatta alcuna assunzione; di conseguenza, tutti i controlli sul tipo dei parametri vengono aboliti. Questo speciale significato attribuito alla lista vuota ha lo scopo di consentire che i programmi scritti con il vecchio C possano essere gestiti correttamente anche dai nuovi compilatori. È comunque bene evitare di usare questa sintassi all'interno dei nuovi programmi. Se la funzione richiede degli argomenti, dichiarateli; se non ne richiede, usate il tipo `void`.

Una volta realizzata `atof`, possiamo riscrivere `atoi` (che converte una stringa nell'intero corrispondente) nel modo seguente:

```
/* atoi: converte la stringa s in un intero, usando atof */
int  atoi(char s[])
{
    double atof(char s[]);

    return (int) atof(s);
}
```

Osservate la struttura delle dichiarazioni e l'istruzione `return`. Il valore dell'espressione in

```
return espressione;
```

viene convertito nel tipo della funzione *prima* che il `return` venga eseguito. Quindi il valore di `atof`, che è un `double`, viene convertito automaticamente in un `int` quando compare in questo `return`, perché la funzione `atoi` ritorna un `int`. Potenzialmente, tuttavia, quest'operazione potrebbe produrre una perdita di informazione; per questo motivo, molti compilatori, incontrandola, la segnalano con un messaggio di warning. La presenza del cast stabilisce esplicitamente che tale conversione è voluta, il che abolisce il messaggio di warning.

Esercizio 4.2 Estendete `atof` in modo che gestisca la notazione scientifica `123.45e-6` in cui un numero decimale può essere seguito da una `e` o una `E` e da un eventuale esponente con segno.

4.3 Variabili Esterne

Un programma in C consiste in un insieme di oggetti esterni, che sono variabili o funzioni. L'aggettivo "esterno" viene usato in opposizione ad "interno", riservato agli argomenti ed alle variabili definite all'interno di qualche funzione. Le variabili esterne sono definite fuori da qualsiasi funzione, e sono perciò a disposizione di più funzioni. Le funzioni stesse, poi, sono sempre esterne, perché il C non consente di definire una funzione allo interno di un'altra. Per default, le variabili e le funzioni esterne godono della proprietà che tutti i riferimenti fatti ad esse tramite lo stesso nome, anche se fatti da funzioni compilate separatamente, si riferiscono allo stesso oggetto (lo standard chiama questa proprietà *linkaggio esterno*). In questo senso, le variabili esterne sono analoghe ai blocchi COMMON del Fortran o alle variabili Pascal dichiarate nel blocco più esterno. Ve-dremo, in seguito, come definire variabili e funzioni esterne visibili all'interno di un unico file sorgente.

Poiché le variabili esterne sono accessibili globalmente, esse costituiscono un modo alternativo di comunicare dati tra le funzioni, diverso dal passaggio di parametri e dai valori di ritorno. Qualsiasi funzione può accedere ad una variabile esterna attraverso il suo nome, purché tale nome sia stato dichiarato.

Se più funzioni devono condividere un elevato numero di variabili, le variabili esterne sono più convenienti rispetto a liste di argomenti molto lunghe. Tuttavia, come abbiamo puntualizzato nel Capitolo 1, questo criterio dovrebbe essere applicato con cautela, perché creare, in un programma, troppe connessioni fra i dati delle diverse funzioni, può risultare dannoso per la sua struttura generale.

Le variabili esterne sono utili anche per il loro vasto scope e per la loro "longevità". Infatti, le variabili automatiche sono interne ad una funzione; esse nascono quando viene iniziata l'esecuzione della funzione, e scompaiono quando essa termina. Al contrario, le variabili esterne sono permanenti, quindi mantengono il valore anche fra due chiamate di funzione. Perciò, se due funzioni che non si invocano a vicenda devono condividere dei dati, è spesso conveniente mantenere tali dati in variabili globali, piuttosto che passarli da una funzione all'altra come argomenti.

Esaminiamo più approfonditamente questo aspetto attraverso un esempio. Vogliamo scrivere un programma di calcolo che gestisca gli operatori `+`, `-`, `*` e `/`. Poiché è più facile da implementare, stabiliamo che il nostro programma usi la notazione Polacca inversa, invece che la notazione infissa (la notazione Polacca inversa è utilizzata da alcune calcolatrici tascabili, oltre che in linguaggi come Forth e Postscript).

In notazione Polacca inversa, ogni operatore segue i suoi operandi; un'espressione in notazione infissa, come

```
(1 - 2) * (4 + 5)
```

diventa cioè

```
1 2 - 4 5 + *
```

Le parentesi non sono necessarie; la notazione non è ambigua, purché si conosca il numero di operandi richiesto da ogni operatore.

L'implementazione è semplice. Ogni operando viene posto in cima ad uno stack; quando si incontra un operatore, dallo stack viene prelevato il numero opportuno di operandi (due per gli operatori binari), ai quali viene applicato l'operatore incontrato, ed il risultato viene nuovamente posto in cima allo stack. Rifacendosi allo esempio precedente, 1 e 2 vengono posti sullo stack; quindi, essi vengono sostituiti con la loro differenza, -1. Poi, sullo stack vengono posti 4 e 5, successivamente rimpiazzati dalla loro somma, 9. Infine, in cima allo stack viene posto il prodotto fra -1 e 9, cioè -9. Al riconoscimento della terminazione dell'input, il valore che si trova in cima allo stack viene prelevato e stampato.

La struttura del programma è quindi un ciclo, che esegue le operazioni opportune su ogni operatore e sui suoi operandi, nell'ordine in cui essi si presentano:

```
while (il prossimo operatore o operando non è EOF)
    if (numero)
        inseriscilo nello stack
    else if (operatore)
        preleva gli operandi
        esegui l'operazione
        inserisci il risultato nello stack
    else if (new line)
        preleva e stampa il valore in cima allo stack
```

```
        else
            errore
```

Le operazioni di inserimento (`push`) e prelevamento (`pop`) dallo stack sono banali ma, poiché gestiscono anche delle condizioni di errore, risultano abbastanza lunghe da suggerire la stesura di funzioni apposite, in modo che non sia necessario ripetere il codice nei diversi punti del programma nei quali tali operazioni sono richieste. Inoltre, è meglio scrivere funzioni apposite anche per la lettura del prossimo operatore od operando in input.

Quello che non è ancora chiaro, a questo punto, è la posizione nella quale inserire lo stack, cioè quali funzioni possano accedervi. Una possibilità consiste nel dichiararlo all'interno del `main`, e passare come argomenti alle funzioni `push` e `pop` lo stack e la posizione corrente al suo interno. Tuttavia, il `main` non ha bisogno di conoscere le variabili che controllano lo stack; esso si limita ad ordinare le operazioni `push` e `pop`. Perciò, abbiamo deciso di memorizzare lo stack e le informazioni ad esso associate in variabili esterne, accessibili alle funzioni `push` e `pop` ma non al `main`.

Codificare queste direttive è relativamente semplice. Se, per ora, supponiamo di inserire tutto il programma in un unico file sorgente, esso apparirà così:

```
#includes
#defines

dichiarazioni di funzione per il main
main()  { .... }

variabili esterne per push e pop

void  push(double f)  { .... }
double  pop(void)  { .... }

int  getop(char s[])  { .... }

funzioni chiamate da getop
```

Nel seguito illustreremo come separare questa struttura su più file sorgenti.

La funzione `main` è un ciclo che contiene un grande `switch` sul tipo di operatore o di operando; quest'uso dello `switch` è più tipico di quello che avevamo mostrato nella Sezione 3.4.

```
#include  <stdio.h>
#include  <math.h>      /* per atof() */

#define  MAXOP  100     /* dimensione massima di operatori ed operandi */
#define  NUMBER  '0'    /* segnala che è stato trovato un numero */

int  getop(char s[]);
void  push(double);
double  pop(void);

/* calcolatrice in notazione Polacca inversa */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type=getop(s))!=EOF)
    {
        switch (type)
        {
            case NUMBER:
                push(atof(s));
                break;
            case '+':
                push(pop()+pop());
```

```
                break;
            case '*':
                push(pop()*pop());
                break;
            case '-':
                op2=pop();
                push(pop()-op2);
                break;
            case '/':
                op2=pop();
                if (op2!=0.0)
                    push(pop()/op2);
                else
                    printf("Errore: divisione per zero\n");
                break;
            case '\n':
                printf("\t%.8g\n", pop());
                break;
            default:
                printf("Errore: comando %s sconosciuto\n", s);
                break;
        }
    }
    return 0;
}
```

Poiché + e * sono operatori commutativi, l'ordine con il quale gli operandi vengono prelevati e combinati fra loro non è rilevante; invece, per gli operatori / e - gli operandi destro e sinistro devono essere distinti. In

```
push(pop()-pop());      /* SBAGLIATO */
```

l'ordine nel quale vengono valutate le due chiamate di `pop` non è definito. Per garantire l'ordine corretto, è necessario prelevare il primo valore e porlo in una variabile temporanea, come abbiamo fatto nel `main`.

```
#define  MAXVAL  100    /* massima profondità dello stack */

int sp=0;               /* prossima posizione libera */
double val[MAXVAL];     /* stack dei valori */

/* push: inserisce f in cima allo stack */
void  push(double f)
{
    if (sp<MAXVAL)
        val[sp++]=f;
    else
        printf("Errore: stack pieno; %g non inseribile\n", f);
}

/* pop: preleva e ritorna il valore in cima allo stack */
double  pop(void)
{
    if (sp>0)
        return val[sp—];
    else
    {
        printf("Errore: lo stack è vuoto\n");
        return 0.0;
    }
}
```

Una variabile è esterna se è stata definita fuori da qualsiasi funzione. Quindi, lo stack ed il suo indice, che devono essere condivisi da `push` e `pop`, sono dichiarati fuori da queste funzioni. Queste dichiarazioni, tuttavia, sono invisibili al `main`, che non fa riferimento allo stack né al suo indice.

Rivolgiamo ora la nostra attenzione all'implementazione di `getop`, la funzione che deve leggere l'operando od operatore successivo. Il suo compito è semplice. Essa deve tralasciare i caratteri di tabulazione e gli

spazi. Se il carattere successivo non è una cifra o un punto decimale, lo restituisce. Altrimenti, raccoglie una strin-ga di cifre (che può comprendere un punto decimale), e ritorna NUMBER, il segnale usato per indicare che è stato letto un numero.

```
#include  <ctype.h>

int  getch(void);
void  ungetch(int);

/* getop: legge il successivo operatore o operando numerico */
int  getop(char s[])
{
    int i, c;

    while ((s[0]=c=getch())==' '  ||  c=='\t')
        ;
    s[1]='\0';
    if (!isdigit(c)  &&  c!='.')
        return c;       /* non è un numero */
    i=0;
    if (isdigit(c))     /* legge la parte intera */
        while (isdigit(s[++i]=c=getch()))
            ;
    if (c=='.')         /* legge la parte frazionaria */
        while (isdigit(s[++i]=c=getch()))
            ;
    s[i]='\0';
    if (c!=EOF)
        ungetch(c);
    return NUMBER;
}
```

Cosa sono getch e ungetch? Spesso accade che un programma non sia in grado di capire se ha letto tutto il suo input prima di avere letto un carattere in più del dovuto. Un esempio classico è quello della lettura dei caratteri che compongono un numero: fino a quando non si incontra il primo carattere diverso da una ci-fra, il numero non è completo. Ma, a questo punto, il programma ha già letto un carattere di troppo, che non è in grado di gestire.
Il problema si risolverebbe se fosse possibile "restituire all'input" il carattere letto in eccesso, dopo averlo esa-minato. Allora, ogni volta che il programma legge un carattere di troppo, potrebbe restituirlo all'input, in modo che il resto del codice possa comportarsi come se quel carattere non fosse mai stato letto. Fortunatamente, un simile comportamento è facilmente simulabile, scrivendo una coppia di funzioni cooperanti tra loro. getch esamina il successivo carattere in input; ungetch memorizza i caratteri rifiutati, in modo che le suc-cessive chiamate a getch li riesamino prima di passare all'input successivo.

Il modo in cui queste due funzioni operano è semplice. ungetch inserisce i caratteri rifiutati in un buffer condiviso: un vettore di caratteri. getch legge da questo buffer e, se esso è vuoto, chiama getchar. È an-che necessario avere un indice che registri la posizione del carattere corrente all'interno del buffer.

Poiché il buffer e l'indice sono condivisi da getch e ungetch, e devono conservare i loro contenuti anche attraverso chiamate successive, essi devono essere esterni ad entrambe le funzioni. Perciò, possiamo scri-vere:

```
#define  BUFSIZE  100

char  buf[BUFSIZE];     /* buffer per ungetch */
int  bufp=0;            /* prossima posizione libera in buf[] */

int  getch(void)        /* preleva un carattere (che potrebbe essere stato
                           rifiutato in precedenza) */
{
    return (bufp>0)?buf [--bufp]:getchar();
}

void  ungetch(int c)    /* rimette un carattere nell'input */
{
```

```
        if (bufp>=BUFSIZE)
            printf("ungetch: troppi caratteri\n");
        else
            buf[bufp++]=c;
    }
```

La libreria standard comprende una funzione `ungetc` che permette di restituire all'input un unico carattere; la illustreremo nel Capitolo 7. Per i caratteri rifiutati abbiamo usato un vettore, invece che un singolo caratte-re, per illustrare un approccio più generale.

Esercizio 4.3 Data la struttura di base, è facile estendere il programma della calcolatrice. Aggiungete l'operatore modulo (`%`) e la gestione dei numeri negativi.

Esercizio 4.4 Aggiungete dei comandi per stampare il primo elemento dello stack senza prelevarlo, per dupli-carlo e per prelevare due elementi in una volta. Aggiungete un comando per pulire lo stack.

Esercizio 4.5 Aggiungete l'accesso a funzioni di libreria come `sin`, `exp` e `pow`. Si veda `<math.h>` nella Appendice B, Sezione 4.

Esercizio 4.6 Aggiungete comandi per la gestione delle variabili (la scelta più semplice consiste nel consentire l'uso di 26 variabili con nomi costituiti da una sola lettera). Aggiungete una variabile che contenga l'ultimo valore stampato.

Esercizio 4.7 Scrivete una funzione `ungets(s)`, che restituisca all'input un'intera stringa. `ungets` deve poter accedere a `buf` e `bufp`, o può limitarsi a chiamare `ungetch`?

Esercizio 4.8 Supponete di non dovere mai rifiutare più di un carattere. Modificate di conseguenza `getch` e `ungetch`.

Esercizio 4.9 La nostra versione di `getch` e `ungetch` non gestisce correttamente un `EOF` rifiutato. Decidete quale sarebbe l'azione corretta da eseguire in presenza di un `EOF` rifiutato ed implementatela.

Esercizio 4.10 Un'altra scelta implementativa usa `getline` per leggere un'intera linea di input; in questo caso, `getch` e `ungetch` sono inutili. Modificate il programma calcolatrice in modo che utilizzi questo approccio.

4.4 Regole di Scope

Le funzioni e le variabili esterne che compongono un programma C non hanno bisogno di essere compilate tutte contemporaneamente; il testo sorgente del programma può essere memorizzato in più file sorgente, e funzioni compilate in precedenza possono essere caricate dalle librerie.

A questo riguardo, esistono alcuni interrogativi degni di nota:
- Come devono essere scritte le dichiarazioni, affinché le variabili siano dichiarate correttamente al momento della compilazione?
- Come devono essere sistemate le dichiarazioni, affinché tutti i segmenti di codice siano assemblati correttamente al momento del caricamento del programma?
- Come devono essere organizzate le dichiarazioni, affinché non siano necessarie duplicazioni?
- Come vengono inizializzate le variabili esterne?

Discutiamo alcuni aspetti riorganizzando il programma calcolatrice in diversi file. Da un punto di vista pratico, questo programma è troppo piccolo perché sia conveniente spezzarlo; tuttavia, il suo smembramento su più file consente di illustrare in dettaglio ciò che accadrebbe con un programma di dimensioni maggiori.

Lo *scope* di un nome è la porzione di programma all'interno della quale tale nome può essere usato. Per una variabile automatica, dichiarata all'inizio di una funzione, lo scope è la funzione stessa. Variabili locali aventi lo stesso nome ma dichiarate in funzioni diverse non sono correlate. La stessa cosa vale per i parametri delle funzioni i quali, in ultima analisi, non sono altro che delle variabili locali.

Lo scope di una variabile esterna o di una funzione va dal punto in cui essa è dichiarata al termine del file sorgente in cui si trova. Per esempio, se `main`, `sp`, `val`, `push` e `pop` sono definite in un unico file nell'ordine illustrato in precedenza, cioè:

```
main()  { .... }

int sp=0;
double val[MAXVAL];

void  push(double f)  { .... }

double  pop(void)  { .... }
```

allora le variabili sp e val, tramite il loro nome, possono essere usate da push e pop; non è necessaria alcun'altra dichiarazione. Al contrario, sp e val non sono visibili al main.

D'altro canto, se è necessario riferire una variabile esterna prima che essa sia stata dichiarata, oppure se essa è definita in un file sorgente diverso da quello in cui viene utilizzata, allora è necessaria una dichiarazione di extern.

È importante distinguere tra la *dichiarazione* di una variabile esterna e la sua *definizione*. Una dichiarazione rende note le proprietà di una variabile (principalmente, il suo tipo); una definizione, invece, provoca anche la allocazione di un'area di memoria riservata a quella variabile. Se le linee

```
int sp;
double val[MAXVAL];
```

appaiono all'esterno di qualsiasi funzione, esse *definiscono* le variabili esterne sp e val, provocano all'alloca-zione della memoria necessaria a queste variabili ed infine servono anche come dichiarazioni delle variabili per tutto il resto del file sorgente. Invece, le linee

```
extern int sp;
extern double val[MAXVAL];
```

dichiarano, per il resto del file sorgente, che sp è una variabile di tipo int, e che val è un vettore di double, la cui dimensione è determinata altrove, ma non creano le variabili né riservano loro alcun'area di memoria.

Fra tutti i file che costituiscono il programma sorgente, uno solo deve contenere la *definizione* di una variabile esterna; gli altri file possono contenere soltanto dichiarazioni di extern, che consentono loro di utilizzare la variabile (anche il file che contiene la definizione può, in realtà, contenere delle dichiarazioni). Le dimensioni del vettore devono essere specificate nella definizione, e sono opzionali nelle dichiarazioni di extern.

L'inizializzazione di una variabile esterna si ha soltanto al momento della definizione.

Sebbene questa non sia l'organizzazione ideale per il programma calcolatrice, inseriamo le funzioni push e pop in un unico file sorgente, e definiamo ed inizializziamo le variabili sp e val in un altro. Per potere utilizzare questi due file, sarebbero allora necessarie le seguenti dichiarazioni:

nel file1:

```
    extern int sp;
    extern double val[MAXVAL];

    void push(double f)  { .... }

    double pop(void)  { .... }
```

nel file2:

```
    int sp=0;
    double val[MAXVAL];
```

Poiché, in *file1*, le dichiarazioni di extern si trovano prima e fuori da qualsiasi funzione, esse si applicano a tutte le funzioni presenti in *file1*. Questa stessa organizzazione sarebbe necessaria se, pur avendo funzioni e variabili in un unico file, le dichiarazioni di sp e val fossero poste dopo il loro utilizzo.

4.5 Header File

Dividiamo ora il programma calcolatrice in diversi file sorgente, come accadrebbe realmente se ognuna delle sue componenti fosse notevolmente più grande. La funzione `main` si trova in un file, che chiameremo `main.c`; `push`, `pop` e le loro variabili si trovano in un altro file, `stack.c`; `getop` è in un terzo file, `getop.c`. Infine, `getch` e `ungetch` si trovano in un quarto file, `getch.c`; separiamo queste due funzioni dalle altre perché, in un programma reale, esse verrebbero prelevate da una libreria compilata separatamente.

L'ultimo aspetto da considerare è la definizione e la dichiarazione delle variabili condivise tra i vari file. Per quanto possibile, vogliamo concentrare definizioni e dichiarazioni in un unico punto, in modo che sia più semplice correggerle e modificarle al variare del programma. Perciò, collochiamo questo materiale comune in un *header file*, `calc.h`, che verrà incluso, in caso di necessità, dagli altri file (la linea `#include` viene descritta nella Sezione 4.11). Il programma risultate assume questo aspetto:

```
calc.h:

#define NUMBER '0'
void  push(double);
double  pop(void);
int  getop(char s[]);
int  getch(void);
void  ungetch(int);
```

```
main.c:

#include  <stdio.h>
#include  <math.h>
#include  "calc.h"
#define  MAXOP  100
main()
{
....
}
```

```
getop.c:

#include  <stdio.h>
#include  <ctype.h>
#include  "calc.h"
getop()
{
....
}
```

```
stack.c:

#include  <stdio.h>
#include  "calc.h"
#define  MAXVAL  100
int sp=0;
double val[MAXVAL];
void  push(double)
{
....
}
double  pop(void)
{
....
}
```

```
getch.c:

#include  <stdio.h>
#define  BUFSIZE  100
char buf[BUFSIZE];
int bufp=0;
int  getch(void)
{
....
}
void  ungetch(int)
{
....
}
```

Esiste un compromesso, fra il desiderio di fare sì che ogni file possa accedere soltanto alle informazioni delle quali ha realmente bisogno, e la realtà pratica, secondo la quale è difficile gestire molti header file. Per programmi di dimensioni ridotte, è probabilmente meglio avere un unico header file, contenente tutto ciò che dev'essere condiviso fra qualsiasi parti del programma; questa è la soluzione che abbiamo adottato in questo

caso. Per programma di dimensioni maggiori, tuttavia, è necessario ricorrere ad un'organizzazione più sofisticata, che comprenda diversi header file.

4.6 Variabili STATIC

Le variabili `sp` e `val` del file `stack.c`, e `buf` e `bufp` del file `getch.c`, sono destinate all'uso privato delle funzioni presenti nei rispettivi file sorgente, e nessun'altra entità ha bisogno di accedervi. La dichiarazione di `static`, applicata ad una variabile esterna o ad una funzione, ne limita lo scope del file sorgente nel quale essa si trova. Per esempio, applicare la dichiarazione di `static` a `buf` e `bufp` significa consertirne l'utilizzo da parte delle funzioni `getch` e `ungetch`, impedendo, al tempo stesso, che tali variabili siano visibili agli utilizzatori di queste due funzioni.

Le variabili di tipo static si ottengono anteponendo ad una normale dichiarazione la parola chiave `static`. Se le due funzioni e le due variabili vengono compilate all'interno di un unico file, come in

```
static char buf[BUFSIZE];     /* buffer per ungetch */
static int bufp=0;            /* prossima posizione in buf */

int  getch(void)  { .... }

void  ungetch(int c)  { .... }
```

allora nessun'altra funzione potrà accedere a `buf` e `bufp`, e questi nomi non saranno in conflitto con nomi uguali eventualmente presenti in altri file dello stesso programma. Analogamente, anche le variabili `sp` e `val`, usate da `push` e `pop` per la manipolazione dello stack, possono essere nascoste, anteponendo alle loro dichiarazioni la parola `static`.
Principalmente, la dichiarazione di `static` viene applicata alle variabili esterne, ma essa è lecita anche sulle funzioni. Di solito, i nomi delle funzioni sono globali e visibili all'intero programma. Se però una funzione è dichiarata `static`, il suo nome diventa visibile soltanto all'interno del file nel quale si trova.

La dichiarazione di `static`, infine, può essere applicata anche alle variabili interne. In questo caso, l'effetto di tale dichiarazione è quello di consentire alla variabile di mantenere il proprio valore anche fra due chiamate successive della funzione alla quale appartiene. Cioè, una variabile automatica dichiarata `static` non scompare al termine dell'esecuzione della funzione, ma continua ad esistere anche dopo che la funzione è terminata. Questo significa che le variabili interne dichiarate `static` consentono di riservare memoria privata e permanente per una particolare funzione.

Esercizio 4.11 Modificate `getop` in modo che non abbia bisogno di usare `ungetch`. Suggerimento: usate una variabile interna `static`.

4.7 Variabili REGISTER

Una dichiarazione `register` avvisa il compilatore che la variabile in questione verrà utilizzata frequentemente. L'idea è che le variabili `register` debbano essere collocate nei registri della macchina, il cui impiego può consentire di ottenere programmi più brevi e più veloci. Tuttavia, i compilatori sono liberi di ignorare tale avvertimento.

La dichiarazione `register` ha la forma

```
register int x;
register char c;
```

Essa può essere applicata soltanto alle variabili automatiche ed ai parametri formali di una funzione. In quest'ultimo caso, appare con la seguente sintassi:

```
f(register unsigned m, register long n)
{
    register int i;

    ....
}
```

Nella pratica, le variabili `register` sono soggette a delle restrizioni, che riflettono la realtà dell'hardware sul quale si opera. Soltanto poche variabili di una funzione possono essere mantenute nei registri, e tali variabili possono appartenere ad un ristretto insieme di tipi. Tuttavia, le dichiarazioni `register` in eccesso sono innocue, perché vengono semplicemente ignorate. Infine, bisogna tenere presente che non è possibile conoscere l'indirizzo di una variabile `register` (un argomento, questo, che verrà trattato nel Capitolo 5), indipendentemente dal fatto che essa venga effettivamente mantenuta o meno in un registro. Le restrizioni specifiche sul numero e sul tipo delle variabili `register` dipendono dalla macchina sulla quale si opera.

4.8 Struttura a Blocchi

Il C non è un linguaggio strutturato a blocchi, nell'accezione con la quale questo termine viene utilizzato per il Pascal ed altri linguaggi simili. In C, infatti, le funzioni non possono essere definite all'interno di altre funzioni. D'altro canto, all'interno di una funzione le variabili possono essere definite secondo il criterio della strutturazione a blocchi. La dichiarazione delle variabili (e la loro inizializzazione) può seguire la parentesi graffa sinistra che introduce una *qualsiasi* istruzione composta, e non soltanto la parentesi che indica l'inizio della funzione. Le variabili dichiarate in questo modo nascondono quelle con nomi uguali dichiarate in blocchi più esterni, e continuano ad esistere fino a che non viene raggiunta la parentesi graffa destra che chiude il blocco. Per esempio, in

```
if (n>0)
{
    int i;        /* dichiara una nuova variabile i */

    for (i=0; i<n; i++)
    ....
}
```

lo scope della variabile `i` è il ramo "then" dell'`if`; questa `i` non è correlata ad alcuna `i` all'esterno del blocco. Una variabile automatica, dichiarata ed inizializzata in un blocco, viene inizializzata ogni volta che il blocco entra in esecuzione. Una variabile `static` viene inizializzata soltanto alla prima esecuzione del blocco.

Anche le variabili automatiche ed i parametri formali nascondono le variabili esterne e le funzioni con lo stesso nome. Date le dichiarazioni

```
int x;
int y;

f(double x)
{
    double y;

    ....
}
```

le occorrenze di `x` all'interno della funzione `f` si riferiscono al parametro di tipo `double`, mentre quelle allo esterno fanno riferimento ad un `int`. La stessa cosa accade con la variabile `y`.

Per motivi di chiarezza, è meglio non usare nomi di variabili che nascondono nomi più esterni; la probabilità che si verifichino errori e confusione è troppo elevata.

4.9 Inizializzazione

Fino a questo momento, il problema dell'inizializzazione è stato nominato solo in margine ad altri argomenti, ma mai approfondito. Ora che abbiamo illustrato le diverse classi di memorizzazione, riassumiamo in questo paragrafo alcune delle regole che governano l'inizializzazione delle variabili.

In assenza di un'inizializzazione esplicita, il C garantisce che le variabili esterne e quelle static vengano inizializzate a zero; le variabili automatiche e quelle dichiarate register, invece, hanno valori iniziali indefiniti (sono, cioè, "sporche").

Le variabili scalari possono essere inizializzate al momento della loro definizione, postponendo al loro nome un segno di uguale ed un'espressione:

```
int x=1;
char squote='\';
long day=1000L*60L*60L*24L;          /* msec / giorno */
```

Per le variabili esterne e static, l'inizializzatore dev'essere un'espressione costante; l'inizializzazione viene fatta una volta, concettualmente prima dell'inizio dell'esecuzione del programma. Per le variabili automatiche e register, essa viene effettuata ogni volta che inizia l'esecuzione della funzione o del blocco interessato.

Per quest'ultimo tipo di variabili, l'inizializzatore può non essere una costante, ma può consistere in una qualsiasi espressione nella quale compaiano valori definiti in precedenza ed anche, eventualmente, chiamate di funzione. Per esempio, le inizializzazioni del programma di ricerca binaria, presentato nella Sezione 3.3, potrebbero essere scritte nella forma seguente:

```
int  binsearch(int x, int v[], int n)
{
    int low=0;
    int high=n-1;
    int mid;
    ....
}
```

invece che nella forma

```
int low, high, mid;

low=0;
high=n-1;
```

In ultima analisi, le inizializzazioni delle variabili automatiche non sono altro che un modo più compatto di scrivere degli assegnamenti. La scelta della forma più opportuna dipende fortemente dalla situazione specifica. In generale, noi abbiamo preferito utilizzare assegnamenti espliciti, perché gli inizializzatori inseriti nelle dichiarazioni sono meno visibili e, spesso, sono lontani dal punto in cui le inizializzazioni sono effettivamente necessarie.

Un vettore può essere inizializzato postponendo alla sua dichiarazione una lista di valori iniziali, racchiusi tra parentesi graffe e separati da virgole. Per esempio, per inizializzare un vettore `days`, contenente il numero dei giorni di ogni mese, potremmo scrivere:

```
int days[]={31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Quando la dimensione di un vettore viene omessa, il compilatore calcola la lunghezza del vettore in base al numero di elementi presenti nella lista di inizializzazione, che in questo caso è dodici.

Se una lista di inizializzatori possiede un numero di elementi inferiore a quello specificato dalla dimensione del vettore, gli elementi restanti vengono inizializzati a zero se il vettore è una variabile esterna o static, ed assumono invece valori indefiniti se il vettore è una variabile automatica. Una lista di inizializzazione troppo lunga costituisce un errore. Non è possibile specificare che uno stesso inizializzatore dev'essere ripetuto più volte, né inizializzare un argomento centrale del vettore senza fornire i valori di tutti gli elementi che lo precedono.

I vettori di caratteri rappresentano, nell'ambito delle regole di inizializzazione, un caso particolare; invece che con una lista delimitata da parentesi e composta da elementi separati da virgole, i vettori di caratteri possono essere inizializzati tramite una stringa costante:

```
char pattern[]="ould";
```

è un'abbreviazione della notazione seguente, equivalente ma meno compatta:

```
char pattern={ 'o', 'u', 'l', 'd', '\0' };
```

In questo caso la dimensione del vettore è cinque (quattro caratteri più quello di terminazione, '\0').

4.10 Ricorsione

In C, le funzioni possono essere usate in modo ricorsivo; cioè, una funzione può richiamare se stessa direttamente o indirettamente. Consideriamo, per esempio, la stampa di un numero sotto forma di stringa di caratteri. Come abbiamo già detto in precedenza, le cifre vengono generate in ordine inverso: quelle meno significative vengono generate per prime, anche se devono essere stampate per ultime.

Questo problema può essere risolto in due modi. Da un lato, le cifre possono essere memorizzate in un vettore nell'ordine in cui vengono generate, ed in seguito questo vettore può essere stampato da destra a sinistra (questo è il metodo adottato nella funzione `itoa` della Sezione 3.6). L'altra soluzione possibile è quella ricorsiva, nella quale la funzione `printd` dapprima chiama se stessa per gestire ogni cifra che incontra, quindi stampa la cifra successiva. Notiamo che anche questa versione può non funzionare con il numero negativo di modulo massimo.

```
#include  <stdio.h>

/* printd: stampa n in decimale */
void  printd(int n)
{
    if (n<0)
    {
        putchar('-');
        n=-n;
    }
    if (n/10)
        printd(n/10);
    putchar(n%10+'0');
}
```

Quando una funzione si richiama ricorsivamente, ogni chiamata provoca la creazione di un insieme completo delle variabili automatiche, nuovo ed indipendente dall'insieme precedente. Per esempio, in `printd(123)` la prima `printd` riceve come argomento `n=123`. Questa chiamata passa l'argomento `n=12` alla seconda, che a sua volta passa alla terza `n=1`. La `printd` di terzo livello stampa `1` e torna al secondo livello. Questa `printd` stampa `2`, e torna alla `printd` di primo livello, che stampa `3` e termina.

Un altro buon esempio di ricorsione è dato dal quicksort, un algoritmo di ordinamento ideato da C. A. R. Hoare nel 1962. Dato un vettore, ne viene scelto un elemento ed il vettore viene suddiviso in due parti: gli elementi minori di quello prescelto e quelli maggiori o uguali. Lo stesso procedimento viene poi applicato, ricorsivamente, ad ognuno dei due sottoinsiemi così ottenuti. Quanto un sottoinsieme risulta composto da meno di due elementi, esso non deve più essere ordinato; questa condizione conclude la ricorsione.

La nostra versione di quicksort non è la più veloce possibile, ma è una delle più semplici. Per suddividere ogni sottoinsieme, utilizziamo l'elemento centrale.

```
/* quicksort: ordina v[left]...v[right] in ordine crescente */
void  qsort(int v[], int left, int right)
{
    int i, last;
    void  swap(int v[], int i, int j);

    if (left>=right)    /* se il vettore contiene meno di */
        return;         /* due elementi, non fa niente    */

    /* sposta l'elemento discriminante in v[left] */
    swap(v, left, (left+right)/2);
    last=left;
    for (i=left+1; i<=right; i++)        /* suddivide */
        if (v[i]<v[left])
            swap(v, ++last, i);

    /* ripristina l'elemento discriminante */
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Poiché l'operazione di scambio, in qsort, ricorre tre volte, abbiamo preferito effettuarla nella funzione swap, scritta appositamente.

```
/* swap: scambia v[i] con v[j] */
void  swap(int v[], int i, int j)
{
    int temp;

    temp=v[i];
    v[i]=v[j];
    v[j]=temp;
}
```

La libreria standard comprende una versione di qsort in grado di ordinare oggetti di qualsiasi tipo.

La ricorsione richiede l'impiego di notevoli quantità di memoria, perché è necessario mantenere uno stack dei valori da processare. Essa, inoltre, difficilmente risulta particolarmente veloce. Tuttavia, il codice ricorsivo è più compatto e, spesso, più facile sia da scrivere che da comprendere rispetto al codice non-ricorsivo equivalente. La ricorsione è particolarmente conveniente per la gestione di strutture dati definite ricorsivamente quali, per esempio, gli alberi; a questo proposito, nella Sezione 6.5 analizzeremo un esempio significativo.

Esercizio 4.12 Sfruttate le idee della funzione printd per scrivere una versione ricorsiva di itoa; in altre parole, convertire un intero in una stringa utilizzando una funzione ricorsiva.

Esercizio 4.13 Scrivete una versione ricorsiva della funzione reverse(s), che inverte una stringa s.

4.11 Il Preprocessore C

Il C fornisce alcune particolari funzionalità del linguaggio per mezzo di un preprocessore che, concettualmente, costituisce la prima fase, separata, della compilazione. Le due funzionalità più utilizzate sono #include, per includere, durante una compilazione, i contenuti di un particolare file, e #define, per sostituire ad un identificatore una stringa arbitraria di caratteri. Altre funzionalità, descritte in questo paragrafo, sono la compilazione condizionale e le macro dotate di argomenti.

4.11.1 Inclusione di File

L'inclusione di file facilita, fra l'altro, la gestione di insiemi complessi di #define e di dichiarazioni. Ogni linea di codice nella forma

```
#include  "nome-file"
```

o

```
#include  <nome-file>
```

viene sostituita con il contenuto del file *nome-file*. Se *nome-file* è racchiuso fra apici, la ricerca del file da includere inizia, normalmente, dalla directory nella quale è stato trovato il programma sorgente; se il file *nome-file* non viene trovato in questa directory, oppure se è racchiuso fra < e >, la ricerca prosegue secondo regole dipendenti dall'implementazione. Un file incluso può a sua volta contenere delle linee del tipo #include.

Spesso, all'inizio di un file, sono presenti diverse linee nella forma #include, che hanno lo scopo di includere istruzioni di #define e dichiarazioni extern comuni, oppure quello di accedere a prototipi di funzione dichiarati in alcuni header, quali <stdio.h> (in realtà, questi ultimi non devono necessariamente essere dei file; vedremo in seguito che i dettagli dell'accesso agli header dipendono dall'implementazione).

L'istruzione #include rappresenta il modo migliore per raggruppare tutte le dichiarazioni relative ad un programma di dimensioni ragguardevoli. Quest'istruzione garantisce che tutti i file sorgente del programma abbiano definizioni e dichiarazioni di variabili coerenti, e questo consente di eliminare una classe di errori molto difficili da individuare. Naturalmente, quando un file incluso viene modificato, tutti i file sorgente che lo includono devono essere ricompilati.

4.11.2 Sostituzione delle Macro

Una definizione ha la forma

```
#define  nome  testo da sostituire
```

Essa richiede una sostituzione di macro del tipo più semplice: tutte le successive occorrenze di *nome* devono essere sostituite con il *testo da sostituire*. Il nome in una `#define` ha la stessa forma di un nome di variabile; il testo da sostituire è, invece, arbitrario. Normalmente, il testo da sostituire occupa la parte restante della linea, ma testi molto lunghi possono proseguire su più linee, purché ognuna di esse, ad eccezione dell'ultima, sia terminata dal carattere `\`. Lo scope di un nome definito con una `#define` va dal punto in cui la `#define` si trova fino al termine del file sorgente. Una definizione può sfruttarne altre date in precedenza. Le sostituzioni vengono effettuate soltanto sui token, e non hanno luogo in caso di stringhe racchiuse fra apici. Per esempio, se YES è un nome definito, la sua occorrenza in `printf("YES")` o in `YESMAN` non comporta alcuna sostituzione.
Qualsiasi nome può essere sostituito con qualsiasi testo. Per esempio,

```
#define  forever  for(;;)          /* ciclo infinito */
```

definisce un nuovo nome, `forever`, che indica un ciclo infinito.

È anche possibile definire macro con argomenti, in modo che il testo da sostituire dipenda dai parametri delle diverse chiamate. Come esempio, definiamo una macro che chiamiamo `max`:

```
#define  max(A, B)  ((A)>(B)?(A):(B))
```

Anche se assomiglia molto ad una chiamata di funzione, ogni invocazione della macro `max` viene espansa in codice in linea. Ogni occorrenza di un parametro formale (`A` e `B` nel nostro esempio) viene rimpiazzata con il corrispondente argomento reale. Perciò la linea

```
x=max(p+q, r+s);
```

viene sostituita con la linea

```
x=((p+q)>(r+s)?(p+q):(r+s));
```

Purché gli argomenti vengano trattati in modo consistente, questa macro può essere utilizzata per ogni tipo di dati; diversamente da quanto accade per le funzioni, non è necessario avere macro diverse per diversi tipi di dati.

Esaminando l'espansione di `max`, noterete alcune particolarità. In primo luogo, le espressioni vengono valutate due volte; se esse hanno degli effetti collaterali, quali l'incremento di variabili o l'input / output, la loro doppia valutazione può creare problemi. Per esempio,

```
max(i++, j++)           /* SBAGLIATO */
```

incrementa due volte la variabile massima fra `i` e `j`. Inoltre, nelle macro con argomenti è necessario porre un'attenzione particolare nell'uso delle parentesi, in modo da essere certi che l'ordine di valutazione sia quello desiderato; analizzate ciò che accade quando la macro

```
#define  square(x)  x*x       /* SBAGLIATO */
```

viene invocata nella forma `square(z+1)`.

Infine, notiamo che anche le macro sono oggetti preziosi. A titolo di esempio, consideriamo l'header `<stdio.h>`, nel quale `getchar` e `putchar` sono spesso definite come macro al fine di evitare, durante la esecuzione, l'overhead che deriverebbe dalla gestione di una chiamata di funzione per ogni carattere trattato. Analogamente, anche le funzioni contenute in `<ctype.h>` sono implementate come macro.

La definizione di un nome può essere annullata con l'istruzione `#undef`; essa, normalmente, viene utilizzata per assicurarsi con una funzione sia definita come tale, piuttosto che come macro:

```
#undef  getchar

int  getchar(void)  { .... }
```

I parametri formali all'interno di stringhe tra apici non vengono sostituiti. Tuttavia se, nel testo da sostituire, un parametro formale è preceduto da un #, la combinazione viene espansa in una stringa tra apici nella quale il parametro è stato rimpiazzato dall'argomento reale. Questa funzionalità può essere utilizzata, per esempio, per costruire macro per la stampa di stringhe di debugging:

```
#define  dprint(expr)  printf(#expr " = %g\n", expr)
```

Quando questa macro viene invocata, come in

```
dprint(x/y);
```

essa viene espansa in

```
printf("x/y" " = %g\n", x/y);
```

All'interno dell'argomento reale, ogni carattere " viene sostituito con \" ed ogni \ con \\, in modo che il risultato sia una stringa costante lecita.

L'operatore di preprocessor ## consente di concatenare argomenti reali durante l'espansione di una macro. Se un parametro del testo da sostituire è preceduto da ##, il parametro viene rimpiazzato con l'argomento reale, l'operatore ## e gli spazi che lo circondano vengono rimossi, ed il risultato viene analizzato nuovamente. Per esempio, la macro `paste` concatena i suoi due argomenti:

```
#define  paste(front, back)  front ## back
```

così `paste(name, 1)` crea la stringa `name1`.

Le regole per l'uso innestato di ## sono complesse; ulteriori dettagli in proposito sono riportati nell'Appendice A.

Esercizio 4.14 Definite una macro `swap(t, x, y)` che scambia fra loro due argomenti di tipo `t` (la struttura a blocchi vi potrà essere utile).

4.11.3 Inclusione Condizionale

Durante la fase di preprocessing, è possibile controllare il preprocessing stesso attraverso delle istruzioni par-ticolari. Esse, infatti, consentono di inserire segmenti di codice molto selettivo, dipendente dal valore di parti-colari condizioni, valutate durante la compilazione.

L'istruzione `#if` valuta un'espressione costante intera (che non può includere costanti di tipo `enum`, operatori di cast e `sizeof`). Se l'espressione è diversa da zero, le linee comprese fra `#if` ed il successivo `#endif`, o `#elif` o `#else` vengono incluse (l'istruzione di preprocessor `#elif` equivale ad un `else if`). L'espressione `defined` (*name*) all'interno di un `#if` assume valore 1 se *name* è stato definito, 0 altrimenti.

Per esempio, per essere certi che il contenuto dell'header `hdr.h` venga incluso una sola volta, possiamo rac-chiudere tale contenuto all'interno di un'espressione condizionale come la seguente:

```
#if !defined(HDR)
#define  HDR

/* contenuto dell'header hdr.h */

#endif
```

La prima inclusione di `hdr.h` definisce il nome `HDR`; eventuali inclusioni successive, trovando `HDR` già definito, salteranno direttamente alla linea `#endif`. Un accorgimento simile a questo può essere utilizzato per evitare di includere un file più volte; se usato in modo consistente, esso consente di fare sì che ogni header includa tutti gli altri header dai quali dipende, senza che l'utente finale si renda conto delle interdipendenze esistenti.

Nell'esempio eseguente, il nome `SYSTEM` viene controllato, al fine di decidere quale versione di header includere:

```
#if SYSTEM==SYSV
    #define  HDR  "sysv.h"
#elif SYSTEM==BSD
    #define  HDR  "bsd.h"
#elif SYSTEM==MSDOS
    #define  HDR  "msdos.h"
#else
    #define  HDR  "default.h"
#endif
#include  HDR
```

Le linee `#ifdef` e `#ifndef` sono delle forme specializzate che controllano se un particolare nome è definito. Nel primo esempio, l'istruzione `#if` avrebbe potuto essere scritta come:

```
#ifndef  HDR
#define  HDR

/* contenuto dell'header hdr.h */

#endif
```

CAPITOLO 5

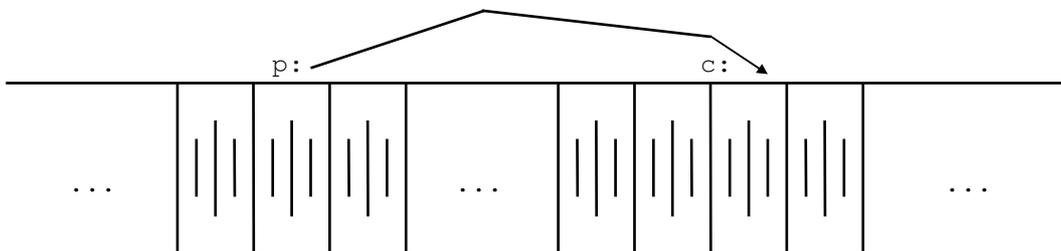
PUNTATORI E VETTORI

Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile. I puntatori, in C, vengono utilizzati molto spesso, in parte perché essi, talvolta, costituiscono l'unico modo possibile di esprimere un calcolo, ed in parte perché, generalmente, consentono di ottenere codice particolarmente compatto ed efficiente. I puntatori ed i vettori sono strettamente correlati; questo capitolo, fra l'altro, evidenzia questa relazione e mostra come sfruttarla.

I puntatori sono stati spesso accomunati all'istruzione `goto` per la loro supposta capacità di generare programmi incomprensibili. Ciò è certamente vero se essi vengono usati distrattamente, ed è facile creare puntatori che puntano a qualcosa di inatteso. Tuttavia, se usati con discernimento, i puntatori possono accrescere la chiarezza e la semplicità del codice. Questa è la loro caratteristica che cercheremo di evidenziare maggiormente.

La variazione principale dell'ANSI C consiste nell'esplicitare le regole che governano la manipolazione dei puntatori, formalizzando, in realtà, ciò che i buoni programmatori ed i buoni compilatori già facevano. Inoltre, nell'ANSI C il tipo `void *` (puntatore ad un `void`) sostituisce il tipo `char *` nella dichiarazione di un puntatore generico.

5.1 Puntatori ed Indirizzi

Iniziamo con una descrizione semplificata dell'organizzazione della memoria. Una macchina, in generale, possiede un vettore di celle di memoria numerate o indirizzate in modo consecutivo; queste celle possono essere manipolate singolarmente o a gruppi. Una situazione tipica è quella in cui un byte viene trattato come un `char`, una coppia di celle di un byte costituisce uno `short`, e quattro byte adiacenti formano un `long`. Un puntatore è un gruppo di celle (spesso due o quattro) che può contenere un indirizzo. Quindi, se `c` è un `char` e `p` un puntatore che punta a `c`, possiamo rappresentare la situazione con lo schema seguente:

L'operatore unario `&` fornisce l'indirizzo di un oggetto, perciò l'istruzione

```
p=&c;
```

assegna l'indirizzo di `c` alla variabile `p`, e si dice che `p` "punta a" `c`. L'operatore `&` si applica soltanto agli oggetti definiti in memoria: le variabili e gli elementi dei vettori. Esso non può essere applicato alle espres-sioni, alle costanti od alle variabili `register`.

L'operatore unario `*` è l'operatore di *indirezione* o *deriferimento*; quando viene applicato ad un puntatore, esso accede all'oggetto puntato. Supponiamo che `x` e `y` siano interi e che `p` sia un puntatore ad un intero. Questa sequenza illustra come dichiarare un puntatore e come utilizzare gli operatori `&` e `*`:

```
int x=1, y=2, z[10];
int *ip;                      /* ip è un puntatore ad un intero */

ip=&x;                        /* ora ip punta a x */
y=*ip;                        /* ora y vale 1 */
*ip=0;                        /* ora x vale 0 */
ip=&z[0];                     /* ora ip punta a z[0] */
```

Le dichiarazioni di x, y e z sono quelle che abbiamo sempre visto. La dichiarazione del puntatore ip,

```
int *ip;
```

afferma che l'espressione *ip è un int. La sintassi della dichiarazione di una variabile rispecchia quella delle espressioni nelle quali la variabile potrebbe comparire. Questo stesso ragionamento si applica alle dichiarazioni di funzione. Per esempio,

```
double *dp, atof(char *);
```

afferma che, in un'espressione, *dp e atof(s) assumono valori di tipo double, e che l'argomento di atof è un puntatore a char.

Per inciso, notate che un puntatore è vincolato a puntare ad un particolare tipo di oggetto: ogni puntatore punta ad uno specifico tipo di dati (esiste un'eccezione: un "puntatore a void" viene usato per supportare qualsiasi tipo di puntatore, ma non può essere referenziato. Torneremo su questo argomento nella Sezione 5.11).

Se ip punta ad un intero x, allora *ip può ricorrere in qualsiasi contesto nel quale può comparire x, quindi

```
*ip=*ip+10;
```

incrementa *ip di 10.

Gli operatori unari * e & sono più vincolanti degli operatori aritmetici, quindi l'assegnamento

```
y=*ip+1;
```

prende ciò che è puntato da ip, qualsiasi cosa sia, lo incrementa di 1 ed assegna il risultato a y, mentre

```
*ip+=1;
```

incrementa di uno l'oggetto puntato da ip, analogamente a

```
++*ip
```

ed a

```
(*ip)++
```

In quest'ultimo esempio, le parentesi sono necessarie; senza di esse, l'espressione incrementerebbe ip invece dell'oggetto puntato, perché gli operatori unari come * e ++ sono associativi da destra a sinistra.

Infine, poiché i puntatori sono delle variabili, essi possono essere usati senza il deriferimento. Per esempio, se iq è un altro puntatore a int,

```
iq=ip;
```

copia il contenuto di ip in iq, cioè fa puntare iq allo stesso oggetto puntato da ip.

5.2 Puntatori ed Argomenti di Funzione

Poiché il C passa alle funzioni gli argomenti per valore, la funzione chiamata non ha un modo diretto per alterare una variabile nella funzione chiamante. Per esempio, una routine di ordinamento può scambiare fra loro due elementi disordinati usando una funzione swap. Non basta scrivere

```
swap(a, b);
```

dove la funzione swap è definita come

```
void  swap(int x, int y)            /* SBAGLIATO */
{
    int temp;
```

```
        temp=x;
        x=y;
        y=temp;
    }
```

A causa della chiamata per valore, la funzione `swap` non può alterare gli argomenti `a` e `b` nella routine chiamante. La funzione `swap` illustrata sopra altera solamente delle *copie* di `a` e `b`.

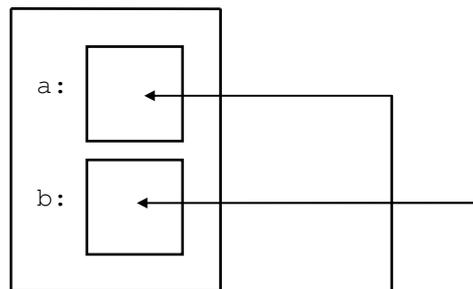
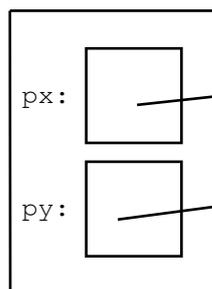
Il modo per ottenere l'effetto desiderato consiste nel passare alla funzione `swap` i *puntatori* agli oggetti che devono essere scambiati:

```
    swap(&a, &b);
```

Poiché l'operatore unario `&` produce l'indirizzo di una variabile, `&a` è un puntatore ad `a`. All'interno della funzione `swap`, i parametri vengono dichiarati come puntatori, attraverso i quali gli operandi vengono acceduti indirettamente.

```
    void  swap(int *px, int *py)        /* scambia *px con *py */
    {
        int temp;

        temp=*px;
        *px=*py;
        *py=temp;
    }
```

Graficamente:

Il fatto che gli argomenti di una funzione siano dichiarati come puntatori consente alla funzione stessa di accedere agli oggetti e di modificarli all'interno della funzione chiamante. Come esempio, consideriamo una funzione `getint`, che esegue una conversione dell'input non formattato, spezzando una sequenza di caratteri in valori interi, un intero per chiamata. `getint` deve restituire il valore che ha trovato, oltre che segnalare quando rileva la fine dell'input mediante il carattere di end of file. Questi valori devono essere restituiti attraverso cammini separati perché altrimenti l'intero usato per `EOF`, qualsiasi sia il suo valore, potrebbe essere interpretato come un intero ricevuto in input.

Una soluzione consiste nel fare in modo che `getint` gestisca l'end of file come valore di ritorno al chiamante, ed utilizzi invece un argomento dichiarato come puntatore per restituire alla funzione chiamante il valore intero convertito. Questo è lo schema impiegato per la funzione `scanf` (si veda, a questo proposi-to, la Sezione 7.4).

Il ciclo seguente riempie un vettore con degli interi ottenuti attraverso successive chiamate a `getint`:

```
int n, array[SIZE], getint(int *);

for (n=0; n<SIZE  &&  getint(&array[n])!=EOF; n++)
    ;
```

Ogni chiamata memorizza in `array[n]` il successivo intero trovato nell'input, ed incrementa `n`. notate che è essenziale passare a `getint` l'indirizzo di `array[n]`. Se ciò non avvenisse, `getint` non avrebbe modo di ritornare al chiamante l'intero convertito.

La nostra versione di `getint` restituisce `EOF` in caso di fine dell'input, zero se l'input successivo non è un numero, ed un valore positivo se l'input contiene un numero.

```
#include  <ctype.h>

int  getch(void);
void  ungetch(int);

/* getint: pone in *pn il prossimo intero letto dall'input */
int  getint(int *pn)
{
    int c, sign;

    while(isspace(c=getch()))       /* evita gli spazi bianchi */
        ;
    if (!isdigit(c)  &&  c!=EOF  &&  c!='+'  && c!='-')
    {
        ungetch(c);
        return 0;
    }
    sign=(c=='-')?-1:1;
    if (c=='+'  ||  c=='-')
        c=getch();
    for (*pn=0; isdigit(c); c=getch())
        *pn=10*(*pn)+(c-'0');
    *pn*=sign;
    if (c!=EOF)
        ungetch(c);
    return c;
}
```

All'interno di `getint`, `*pn` viene usato come una normale variabile intera. Abbiamo anche utilizzato `getch` e `ungetch` (descritte nella Sezione 4.3), in modo che il carattere in eccesso, che deve necessariamente essere letto, possa poi essere ripristinato all'interno dell'input.

Esercizio 5.1 Nella versione data, `getint` considera una valida rappresentazione dello zero un `+` od un `-` non seguiti da alcuna cifra. Modificatela in modo che essa ripristini nell'input un carattere di questo tipo.

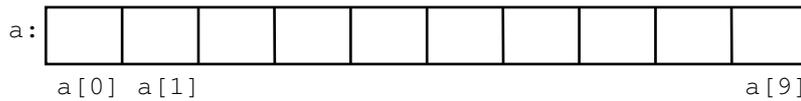
Esercizio 5.2 Scrivete la funzione `getfloat`, versione floating-point della funzione `getint`. Che tipo ritorna la funzione `getfloat` al chiamante?

5.3 Puntatori e Vettori

In C, la relazione esistente tra puntatori e vettori è così stretta da consentirne la trattazione simultanea di questi due argomenti. Qualsiasi operazione effettuabile indicizzando un vettore può essere eseguita anche tramite i puntatori. In generale, la versione che utilizza i puntatori è più veloce ma, almeno per i programmatori inesperti, meno comprensibile.

La dichiarazione

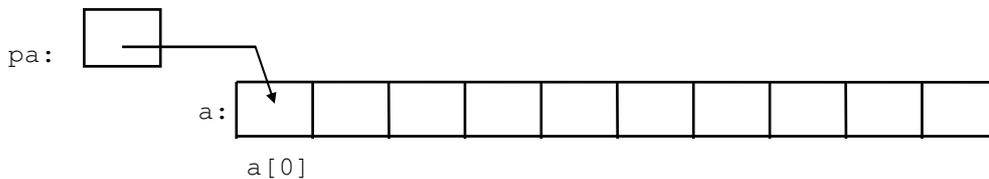
```
int a[10];
```

definisce un vettore a di ampiezza 10, cioè un blocco di dieci oggetti consecutivi, chiamati a[0], a[1], ..., a[9].

La notazione a[i] si riferisce all'i-esimo elemento del vettore. Se pa è un puntatore ad un intero, dichiarato come

```
int *pa;
```

allora l'assegnamento

```
pa=&a[0];
```

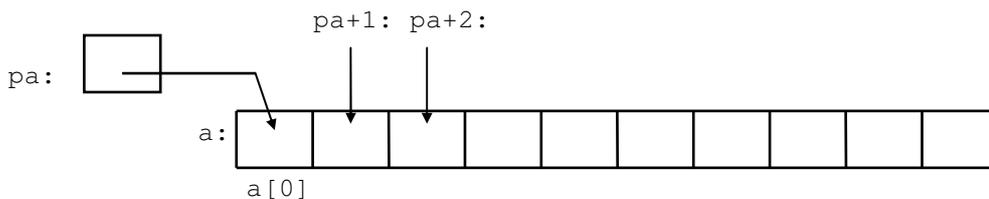
fa in modo che pa punti all'elemento zero di a; cioè, pa contiene l'indirizzo di a[0].

Ora l'assegnamento

```
x=*pa;
```

copierà il contenuto di a[0] in x.

Se pa punta ad un particolare elemento di un vettore, allora, per definizione, pa+1 punta all'elemento successivo, pa+i punta ad i elementi dopo pa, e pa-1 punta ad i elementi prima di pa. Cioè, se pa punta ad a[0],

```
*(pa+1)
```

si riferisce al contenuto di a[1], pa+1 è l'indirizzo di a[i], e *(pa+i) è il contenuto di a[i].

Queste osservazioni sono valide indipendentemente dal tipo o dall'ampiezza delle variabili contenute in a. Il significato della frase "sommare 1 ad un puntatore" e, per estensione, tutta l'aritmetica dei puntatori, è che pa+1 punta all'elemento successivo, e pa+i punta all'i-esimo elemento dopo quello puntato da pa.

La corrispondenza fra indicizzazione di un vettore e aritmetica dei puntatori è molto stretta. Per definizione, il valore di una variabile o di un'espressione di tipo vettore è l'indirizzo dell'elemento zero del vettore stesso. Quindi, dopo l'assegnamento

```
pa=&a[0];
```

pa ed a hanno valori identici. Poiché il nome di un vettore è sinonimo della posizione del suo primo elemento, l'assegnamento pa=&a[0] può essere scritto anche nella forma

```
pa=a;
```

Ancora più sorprendente, almeno ad una prima analisi, è il fatto che un riferimento ad a[i] può essere scritto nella forma *(a+i). Nel valutare a[i], il C converte immediatamente quest'espressione in *(a+i);

le due forme sono del tutto equivalenti. Applicando l'operatore `&` ad entrambi i membri dell'equivalenza, otteniamo che anche `&a[i]` e `(a+i)` sono identici: `(a+i)` è l'indirizzo all'`i`-esimo elemento di `a`. Un'altra conseguenza di tutto ciò è che, se `pa` è un puntatore, nelle espressioni esso può essere usato unitamente ad un indice; `pa[i]` è equivalente a `*(pa+i)`. Riassumendo, possiamo dire che un'espressione sotto for-ma di vettori e indici è equivalente ad una che utilizza puntatori e spiazzamenti (offset).

Tra il nome di un vettore ed un puntatore esiste però una differenza che deve sempre essere tenuta presen-te. Un puntatore è una variabile, quindi espressioni come `pa=a` e `pa++` sono legali. Ma il nome di un vettore non è una variabile; costruzioni come `a=pa` ed `a++` sono illegali.

Quando il nome di un vettore viene passato ad una funzione, ciò che viene passato è la posizione dell'ele-mento iniziale. All'interno della funzione chiamata, questo argomento è una variabile locale, quindi un nome di vettore passato come parametro è in realtà un puntatore, ovvero una variabile che contiene un indirizzo. Possiamo utilizzare quest'osservazione per scrivere una nuova versione della funzione `strlen`, che calco-la la lunghezza di una stringa.

```
/* strlen: restituisce la lunghezza della stringa s */
int  strlen(char *s)
{
    int n;

    for (n=0; *s!='\0'; s++)
        n++;
    return n;
}
```

Poiché `s` è un puntatore, il suo incremento è un'operazione del tutto legale; l'espressione `s++` non ha alcun effetto sulla stringa di caratteri nella funzione chiamante, perché incrementa semplicemente la copia privata di `strlen` del puntatore. Questo significa che chiamate del tipo

```
strlen("Salve, mondo");        /* stringa costante */
strlen(array);                 /* char array[100]  */
strlen(ptr);                   /* char *ptr        */
```

sono tutte legali.

Come parametri formali di una funzione, le due forme

```
char s[];
```

e

```
char *s;
```

sono equivalenti; noi preferiamo utilizzare la seconda, perché evidenzia maggiormente che il parametro è un puntatore. Quando ad una funzione viene passato il nome di un vettore, la funzione può decidere se manipolarlo come un vettore o come un puntatore. Se le sembra appropriato e chiaro, essa può anche decidere di utilizzare entrambe le notazioni.

È anche possibile passare ad una funzione soltanto una parte di un vettore, passandole un puntatore all'inizio del sottovettore. Per esempio, se `a` è un vettore,

```
f(&a[2])
```

e

```
f(a+2)
```

sono due modi di passare alla funzione `f` il sottovettore che inizia da `a[2]`. All'interno di `f`, la dichiarazione del parametro può essere:

```
f(int arr[])  { .... }
```

oppure

```
f(int *arr)  { .... }
```

Per quanto riguarda la funzione `f`, il fatto che il parametro si riferisca soltanto ad una parte di un vettore più ampio non ha alcuna importanza.

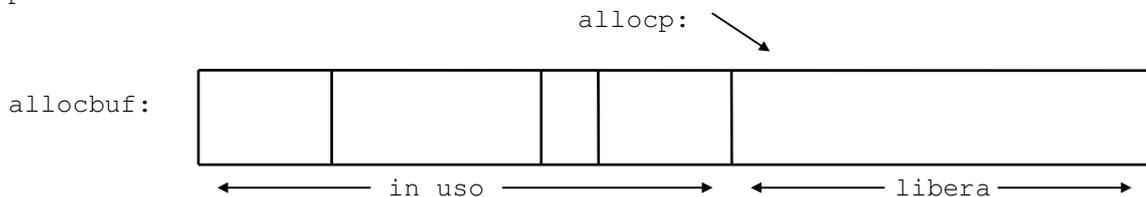
Se si è certi dell'esistenza degli elementi, è anche possibile indirizzare un vettore all'indietro; `p[-1]` e `p[-2]` sono espressioni sintatticamente corrette, e si riferiscono agli elementi che precedono `p[0]`. Natural-mente, non è consentito riferirsi ad oggetti che non sono all'interno degli estremi del vettore.

5.4 Aritmetica degli Indirizzi

Se `p` è un puntatore a qualche elemento di un vettore, allora `p++` incrementa `p` in modo da farlo puntare allo elemento successivo, mentre `p+=i` lo fa puntare ad `i` elementi dopo quello puntato correntemente. Queste ed altre costruzioni simili sono le forme più semplici di aritmetica dei puntatori o degli indirizzi.

Nel suo approccio all'aritmetica degli indirizzi, il C è consistente e regolare; l'integrazione fra puntatori, vettori ed aritmetica degli indirizzi è uno dei punti di forza del linguaggio. Illustriamo questo aspetto scrivendo un rudimentale allocatore di memoria. Ci sono due routine. La prima, `alloc(n)`, restituisce un puntatore `p` ad `n` caratteri consecutivi, che possono essere usati dal chiamante di `alloc` per memorizzare dei caratteri. La seconda routine, `afree(p)`, rilascia la memoria acquisita tramite `alloc`, in modo che possa essere riutilizzata. Le funzioni sono "rudimentali", perché le chiamate ad `afree` devono essere effettuate in ordine oppo-sto a quelle di `alloc`. Cioè, la memoria gestita da `alloc` e `afree` è uno stack, ovvero una lista di tipo last-in, first-out. La libreria standard fornisce funzioni analoghe a quelle chiamate `malloc` e `free` che non hanno queste restrizioni; nella Sezione 8.7 vedremo come esse possono essere implementate.

L'implementazione più semplice consiste nel fare in modo che `alloc` gestisca segmenti di un grande vettore di caratteri, che chiameremo `allocbuf`. Questo vettore è privato ad `alloc` e ad `afree`. Poiché queste due funzioni lavorano sui puntatori, e non sugli indici dei vettori, nessun'altra routine ha bisogno di conoscere il nome del vettori, che può quindi essere dichiarato `static` nel file sorgente che contiene `alloc` ed `afree`, rimanendo perciò invisibile all'esterno. Nell'implementazione pratica, il vettore potrebbe anche non avere alcun nome; esso potrebbe infatti essere ottenuto invocando `malloc`, o chiedendo al sistema operativo un puntatore ad un blocco di memoria privo di nome.

L'altra informazione necessaria è quella sul grado di utilizzo di `allocbuf`. Per questo utilizziamo un puntatore, che chiamiamo `allocp`, che punta al primo elemento libero. Quando ad `alloc` vengono chiesti `n` caratteri, la funzione controlla che in `allocbuf` ci sia spazio sufficiente. Se ciò avviene, `alloc` ritorna il valore corrente di `allocp` (cioè l'inizio del blocco libero), ed incrementa questo puntatore di `n` posizioni, in modo da farlo puntare alla successiva area libera. Se non c'è spazio sufficiente, `alloc` ritorna zero, `afree(p)` si limita ad assegnare ad `allocp` il valore `p`, se esso cade all'interno di `allocbuf`.

```
#define  ALLOCSIZE  10000           /*dim. spazio disponibile */

static char allocbuf[ALLOCSIZE];    /* memoria per alloc */
static char *allocp=allocbuf;       /* prox. posiz. libera */
```

```
char  *alloc(int n)           /* ritorna un puntatore ad n caratteri */
{
    /* se c'è spazio */
    if (allocbuf+ALLOCSIZE-allocp>=n)
    {
        allocp+=n;            /* vecchio puntatore */
        return allocp-n;
    }
    else
        return 0;             /* non c'è spazio */
}

void  afree(char *p)          /* libera la memoria puntata da p */
{
    if (p>=allocbuf  &&  p<allocbuf+ALLOCSIZE)
        allocp=p;
}
```

In generale un puntatore può essere inizializzato come qualsiasi altra variabile, anche se normalmente gli unici valori significativi sono zero, oppure un'espressione che coinvolge gli indirizzi di dati del tipo appro-priato definiti in precedenza. La dichiarazione

```
static char *allocp=allocbuf;
```

definisce `allocp` come un puntatore a carattere e lo inizializza in modo che punti all'inizio di `allocbuf`, che è la prima posizione libera quando l'esecuzione del programma inizia. Avremmo anche potuto scrivere

```
static char *allocp=&allocbuf[0];
```

poiché il nome del vettore è l'indirizzo dell'elemento zero.

Il test

```
if (allocbuf+ALLOCSIZE-allocp>=n)  { /* c'è spazio */ }
```

controlla che esiste spazio sufficiente per soddisfare la richiesta di `n` caratteri. Se lo spazio c'è, il nuovo valore di `allocp` dovrebbe al più superare di un'unità la dimensione di `allocbuf`. Se la richiesta può essere soddisfatta, `alloc` restituisce un puntatore all'inizio di un blocco di caratteri (notate, a questo proposito, la dichiarazione della funzione stessa). In caso contrario, `alloc` deve segnalare il fatto che non c'è spazio in `allocbuf`. Il C garantisce che zero non sia mai un indirizzo valido per i dati, quindi zero, come valore di ritorno, può essere utilizzato per segnalare un evento anomalo come, nel nostro caso, la mancanza di spazio.

I puntatori e gli interi, in generale, non sono interscambiabili. Lo zero costituisce l'unica eccezione: la costante zero può essere assegnata ad un puntatore, ed un puntatore può a sua volta essere confrontato con la costante zero. La costante simbolica `NULL` viene spesso usata al posto dello zero, come nome mne-monico per indicare più chiaramente che questo, per un puntatore, è un valore speciale. La costante simbo-lica `NULL` è definita in `<stdio.h>`. Da questo momento in poi, noi la utilizzeremo sempre al posto dello zero.

Controlli del tipo

```
if (allocbuf+ALLOCSIZE-allocp>=n)  { /* c'è spazio */ }
```

e

```
if (p>=allocbuf  &&  p<allocbuf+ALLOCSIZE)
```

mostrano alcuni importanti aspetti dell'aritmetica dei puntatori. In primo luogo i puntatori, in certe circostan-ze, possono essere confrontati. Se `p` e `q` puntano a membri dello stesso vettore, operatori relazionali come `==`, `!=`, `<`, `>=`, ecc. lavorano correttamente. Per esempio,

```
p<q
```

è vero se p punta ad un membro del vettore che precede quello puntato da q. Qualsiasi puntatore può essere confrontato con lo zero. Tuttavia, il comportamento di operazioni e confronti fra puntatori che non puntano ad elementi di uno stesso vettore è indefinito (esiste un'eccezione: l'indirizzo del primo elemento dopo che la fine di un vettore può essere utilizzato nell'aritmetica dei puntatori).

In secondo luogo, abbiamo già osservato che un puntatore ed un intero possono esser sommati o sottratti. La costruzione

```
p+n
```

indica l'indirizzo dell'n-esimo oggetto che segue quello attualmente puntato da p. Questo è vero indipendentemente dal tipo di oggetto a cui punta p; n viene dimensionato in base alla dimensione degli oggetti ai quali p punta, e tale dimensione è determinata dalla dichiarazione di p stesso. Se un intero occu-pa quattro byte, per esempio, p verrà incrementato di quattro byte in quattro byte.

Anche la sottrazione fra puntatori è un'operazione legale: se p e q puntano ad elementi di uno stesso vettore, e p<q, allora q-p+1 è il numero di elementi fra p e q, estremi inclusi. Quest'osservazione può essere usata per scrivere un'ulteriore versione di strlen:

```
/* strlen: ritorna la lunghezza della stringa s */
int  strlen(char *s)
{
    char *p=s;

    while (*p!='\0')
        p++;
    return p-s;
}
```

In questa dichiarazione, p viene inizializzato ad s, cioè viene fatto puntare al primo carattere della stringa s. Nel ciclo di while, ogni carattere viene confrontato con il carattere nullo, fino a che quest'ultimo non viene raggiunto. Poiché p punta ad un carattere, p++ incrementa p facendolo puntare al carattere successivo, e p-s rappresenta il numero di carattere scanditi, cioè la lunghezza della stringa (il numero di caratteri che compongono la stringa potrebbe essere troppo elevato per poter essere contenuto in un int. L'header <stddef.h> definisce un tipo ptrdiff_t in grado di contenere la differenza, con segno, fra due puntatori. In ogni caso, se fossimo stati precisi, avremmo usato il tipo size_t come tipo del valore di ritorno di strlen, per coerenza con la libreria standard. size_t è il tipo unsigned int, ritornato dall'operatore sizeof).

L'aritmetica dei puntatori è consistente: se avessimo utilizzato dei float, che occupano più memoria dei char, e se p fosse stato un puntatore ad oggetti di tipo float, p++ avrebbe spostato p sul float successivo. Quindi, noi potremmo scrivere una nuova versione di alloc che memorizza oggetti float invece che char, e per farlo sarebbe sufficiente sostituire i char con i float all'interno di alloc e di afree. Tutte le operazioni sui puntatori terrebbero automaticamente conto della nuova dimensioni degli oggetti trattati.

Le operazioni consentite sui puntatori sono l'assegnamento fra puntatori dello stesso tipo, l'addizione e sottrazione fra puntatori ed interi, la sottrazione ed il confronto fra due puntatori ad elementi di uno stesso vettore, e l'assegnamento ed il confronto con lo zero. Tutte le altre operazioni aritmetiche sui puntatori sono illegali. È illegale sommare fra loro due puntatori, moltiplicarli, dividerli od effettuare su di essi degli shift; è illegale applicare loro gli operatori bit a bit e sommare loro quantità float o double; infine, fatta eccezione per il tipo void *, è illegale assegnare ad un puntatore di un tipo un puntatore di tipo diverso senza utilizzare l'operatore di cast.

5.5 Puntatori a Caratteri e Funzioni

Una *stringa costante*, scritta come

```
"Io sono una stringa"
```

è un vettore di caratteri. Nella rappresentazione interna, il vettore è terminato dal carattere nullo '\0', in modo che il programma possa trovarne la fine. La lunghezza della stringa in memoria supera quindi di una unità il numero dei caratteri compresi fra i doppi apici.

Probabilmente, il caso più comune di occorrenza di stringhe costanti è quello del passaggio di parametri alle funzioni, come in

```
printf("Salve, mondo\n");
```

Quando, in un programma, compare una stringa di caratteri come questa, l'accesso ad essa avviene attraverso un puntatore a carattere; `printf` riceve un puntatore all'inizio del vettore di caratteri. In altre parole, una stringa costante viene acceduta tramite un puntatore al suo primo elemento.

Le stringhe costanti possono anche non essere degli argomenti di funzione. Se `pmessage` è dichiarato come

```
char *pmessage;
```

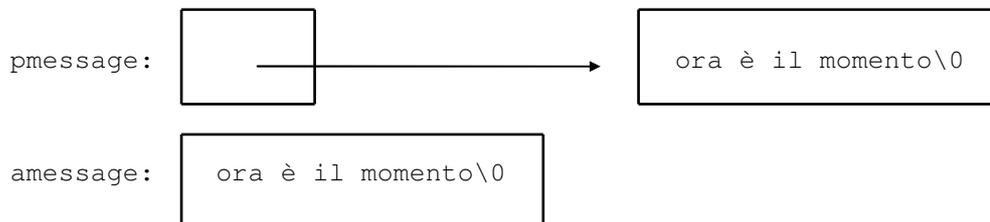
allora l'istruzione

```
pmessage="ora è il momento";
```

assegna a `pmessage` un puntatore al vettore di caratteri. Notate che *non* viene fatta alcuna copia della stringa; in quest'operazione sono coinvolti soltanto dei puntatori. Il C non fornisce alcun operatore che consenta di trattare una stringa come un'entità unica.

Tra le seguenti definizioni esiste un'importante differenza:

```
char amessage[]="ora è il momento";      /* un vettore */
char *pmessage="ora è il momento";       /* un puntatore */
```

`amessage` è un vettore, sufficientemente grande da contenere la sequenza di caratteri e lo `'\0'` che lo inizializzano. I singoli caratteri all'interno del vettore possono cambiare, ma `amessage` si riferisce sempre alla stessa area di memoria. Al contrario, `pmessage` è un puntatore, inizializzato in modo che punti ad una stringa costante; di conseguenza, esso può essere modificato in modo che punti altrove, ma se tentate di modificare il contenuto della stringa il risultato sarà indefinito.

Ora illustreremo altri aspetti dei puntatori e dei vettori, studiando diverse versioni di due utili funzioni adattate alla libreria standard. La prima funzione è `strcpy(s, t)`, che copia la stringa `t` nella stringa `s`. Sarebbe bello potere scrivere semplicemente `s=t`, ma questo copia soltanto i puntatori, non i caratteri. Per copiare i caratteri, abbiamo bisogno di un ciclo. La prima versione usa i vettori:

```
/* strcpy: copia t in s; versione con i vettori */
void  strcpy(char *s, char *t)
{
    int i;

    i=0;
    while ((s[i]=t[i])!='\0')
        i++;
}
```

Per confronto, scriviamo una versione di `strcpy` che utilizza i puntatori:

```
/* strcpy: copia t in s; prima versione con i puntatori */
void  strcpy(char *s, char *t)
{
    while ((*s=*t)!='\0')
    {
```

```
        s++;
        t++;
    }
}
```

Poiché gli argomenti vengono passati per valore, `strcpy` può utilizzare i parametri `s` e `t` come meglio crede. Nel nostro caso essi sono dei puntatori inizializzati nel modo opportuno, che vengono spostati lungo i vettori al ritmo di un carattere per volta, fino a che il carattere `'\0'` che termina `t` non è copiato in `s`.

In realtà, la funzione `strcpy` non dovrebbe essere scritta come abbiamo mostrato nella versione precedente. Dei programmatori esperti preferirebbero la forma seguente:

```
/* strcpy: copia t in s; seconda versione con i puntatori */
void  strcpy(char *s, char *t)
{
    while((*s++=*t++)!='\0')
        ;
}
```

Questa versione sposta l'incremento di `s` e di `t` nella parte di controllo del ciclo. Il valore di `*t++` è il carattere puntato da `t` prima dell'incremento; l'operatore `++` postfisso non altera `t` fino a quando questo carattere non è stato trattato. Analogamente, il carattere viene memorizzato nella vecchia posizione puntata da `s`, prima che `s` venga incrementato. Questo carattere è anche il valore che viene confrontato con `'\0'`, per controllare il ciclo. L'effetto finale è che i caratteri vengono copiati da `t` a `s`, fino allo `'\0'` incluso.

Come abbreviazione finale, osserviamo che un confronto con `'\0'` è ridondante, perché in realtà ciò che vogliamo sapere è se l'espressione ha un valore diverso da zero. Quindi, la funzione `strcpy` potrebbe essere scritta nella forma

```
/* strcpy: copia t in s; terza versione con i puntatori */
void  strcpy(char *s, char *t)
{
    while (*s++=*t++)
        ;
}
```

Anche questa versione, ad una prima analisi, può sembrare poco chiara, la convenzione notazionale che ne deriva è considerevole, e questo idioma dovrebbe essere compreso a fondo, perché è molto frequente all'interno di programmi C. La funzione `strcpy` fornita dalla libreria standard (`<string.h>`) restituisce, come valore di ritorno della funzione, la stringa `s` modificata.

La seconda routine che esamineremo è `strcmp(s, t)`, che confronta le due stringhe `s` e `t`, e restituisce un valore negativo, nullo o positivo se `s` è lessicograficamente minore, uguale o maggiore di `t`. Il valore viene ottenuto sottraendo fra loro i caratteri della prima posizione nella quale `s` e `t` differiscono.

```
/* strcmp: restituisce <0 se s<t, 0 se s==t, >0 se s>t */
int  strcmp(char *s, char *t)
{
    int i;

    for (i=0; s[i]==t[i]; i++)
        if (s[i]=='\0')
            return 0;
    return s[i]-t[i];
}
```

La versione di `strcmp` che usa i puntatori è la seguente:

```
/* strcmp: restituisce <0 se s<t, 0 se s==t, >0 se s>t */
int  strcmp(char *s, char *t)
{
    for ( ; *s==*t; s++, t++)
        if (*s=='\0')
            return 0;
```

```
    return *s-*t;
}
```

Poiché ++ e -- possono essere operatori sia prefissi che postfissi, è possibile trovare, anche se più raramente, combinazioni diverse di *, ++ e --. Per esempio,

```
*--p
```

decrementa p prima di prelevare il carattere al quale p punta. In realtà, la coppia di espressioni

```
*p++=val;          /* mette val in cima allo stack */
val=*--p;          /* preleva l'elemento in cima allo stack
                      e lo assegna a val */
```

sono gli idiomi standard per inserire e prelevare elementi da uno stack; si veda a questo proposito la Sezione 4.3.

L'header <string.h> contiene le dichiarazioni relative alle funzioni trattate in questa sezione, oltre ad un vasto insieme di funzioni della libreria standard dedicate alla gestione delle stringhe.

Esercizio 5.3 Scrivete una versione della funzione strcat (illustrata nel Capitolo 2) che utilizzi i puntatori; strcat(s, t) copia la stringa t al termine della stringa s.

Esercizio 5.4 Scrivete la funzione strend(s, t) che ritorna 1 se la stringa t compare al termine della stringa s, 0 altrimenti.

Esercizio 5.5 Scrivete delle funzioni di libreria strncpy, strncat e strncmp, che operano al più su n caratteri delle stringhe passate loro come argomenti. Per esempio, strncpy(s, t, n) copia al più n caratteri di t in s. Le descrizioni complete sono nell'Appendice B.

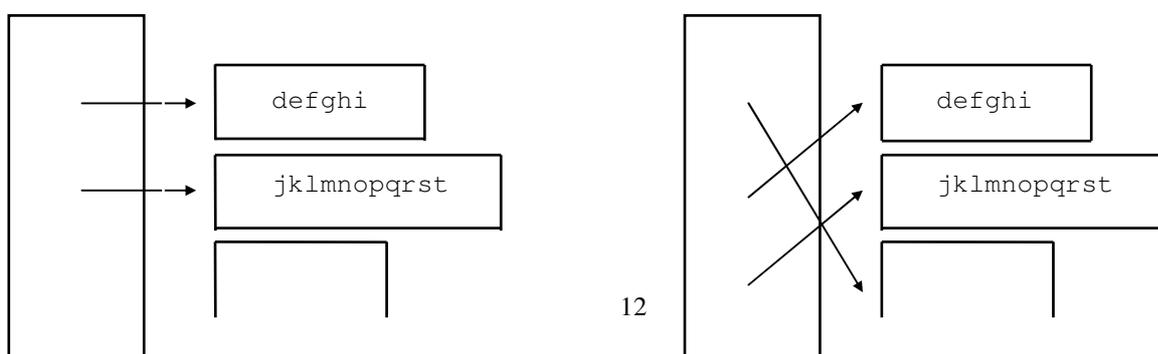
Esercizio 5.6 Riscrivete i programmi dei capitoli e degli esercizi precedenti utilizzando puntatori invece che vettori. Riscrivete, per esempio, getline (Capitoli 1 e 4), atoi, itoa e le loro varianti (Capitoli 2, 3 e 4), reverse (Capitolo 3), strindex e getop (Capitolo 4).

5.6 Vettori di Puntatori e Puntatori a Puntatori

Poiché i puntatori sono a loro volta delle variabili, essi possono essere memorizzati esattamente come delle variabili qualsiasi. Illustriamo questo aspetto scrivendo un programma che ordina alfabeticamente un insieme di linee di testo, una versione semplificata del programma UNIX sort.

Nel Capitolo 3 abbiamo presentato una funzione che ordinava un vettore di interi utilizzando l'algoritmo di Shell sort, e nel Capitolo 4 l'abbiamo migliorata usando l'algoritmo quicksort. Questi stessi algoritmi sono ancora utilizzabili, solo che ora dovranno operare su linee di testo di lunghezze diverse le quali, a diffe-renza degli interi, non possono essere confrontate o spostate con una singola operazione. Abbiamo biso-gno di una forma di rappresentazione dei dati che ci consenta di trattate in modo corretto ed efficiente linee di testo di lunghezza variabile.

Questo è il punto nel quale si inseriscono i vettori di puntatori. Se le linee da ordinare venissero memo-rizzate l'una dopo l'altra in un vettore di caratteri molto lungo, ogni linea potrebbe essere individuata tramite un puntatore al suo primo carattere. Inoltre anche i puntatori possono essere memorizzati in un vettore. Due linee possono essere confrontate passando alla funzione strcmp i loro puntatori come argomenti. Quando due linee, non rispettano l'ordine desiderato, devono essere scambiate, ciò che viene spostato sono i loro indirizzi nel vettore di puntatori, e non le linee stesse.

```
→   abc                                   abc
    ___________                           ___________
```

Questa struttura elimina il duplice problema della complessa gestione della memoria e dell'elevato overhead che si avrebbe se si dovessero spostare le linee di testo.

Il processo di ordinamento è suddiviso in tre fasi:

```
leggi tutte le linee di input
ordinale
stampale in ordine
```

Come al solito, è meglio dividere il programma in funzioni che rispecchiano la naturale suddivisione del problema. Per il momento, rinviamo la trattazione della fase di ordinamento, e concentriamoci sulle strutture dati, sull'input e sull'output.

La routine di input deve leggere e memorizzare i caratteri di ogni linea, e costituire un vettore di puntatori alle linee. Essa deve anche contare il numero delle linee di input, poiché quest'informazione è necessaria per lo ordinamento e la stampa. Dal momento che la funzione di input può trattare soltanto un numero finito di linee, essa può restituire un valore particolare, per esempio -1, se le linee inserite sono troppe.

La routine di output deve solamente stampare le linee dell'ordine in cui compaiono all'interno del vettore di puntatori.

```
#include  <stdio.h>
#include  <string.h>

#define  MAXLINES  5000        /* numero massimo di linee */

char  *lineptr[MAXLINES];      /* puntatore alle linee di testo */

int  readlines(char *lineptr[], int nlines);
void  writelines(char *lineptr[], int nlines);

void  qsort(char *lineptr[], int left, int right);

/* ordina le linee di input */
main()
{
    int nlines;                /* numero di linee di input lette */

    if ((nlines=readlines(lineptr, MAXLINES))>=0)
    {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    }
    else
    {
        printf("Errore: troppe linee di input da ordinare\n");
        return 1;
    }

#define  MAXLEN  1000          /* lunghezza massima di una linea */
int  getline(char *, int);
char  *alloc(int);

/* readlines: legge le linee di input */
int  readlines(char *lineptr[], int maxlines);
{
    int len, nlines;
    char *p, line[MAXLEN];

    nlines=0;
```

```
    while ((len=getline(line, MAXLEN))>0)
        if (nlines>=maxines  ||  (p=alloc(len))==NULL)
            return -1;
        else
        {
            line[len-1]='\0';       /* elimina il new line */
            strcpy(p, line);
            lineptr[nlines++]=p;
        }
    return nlines;
}

/* writelines: scrive in output le linee */
void  writelines(char *lineptr[], int nlines)
{
    int i;

    for (i=0; i<nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

La funzione `getline` è quella presentata nella Sezione 1.9.
La novità principale è la dichiarazione di `lineptr`:

```
char *lineptr[MAXLINES];
```

essa afferma che `lineptr` è un vettore `MAXLINES` elementi, ognuno dei quali è un puntatore a carattere. In altre parole, `lineptr[i]` è un puntatore a carattere, e `*lineptr[i]` è il carattere al quale punta, cioè il primo carattere dell'`i`-esima linea di testo salvata.

Poiché `lineptr` è a sua volta il nome di un vettore, esso può essere trattato come puntatore, analogamente a quanto avveniva nei nostri precedenti esempi, e la funzione `writelines` può essere riscritta nella forma seguente:

```
/* writelines: scrive in output le linee */
void  writelines(char *lineptr[], int nlines)
{
    while (nlines-->0)
        printf("%s\n", *lineptr++);
}
```

Inizialmente `*lineptr` punta alla prima linea; ogni incremento successivo lo fa spostare sul puntatore alla linea successiva, mano a mano che `nlines` si decrementa.

Una volta scritte le routine di gestione dell'input e dell'output, possiamo dedicarci all'ordinamento. La funzione quicksort scritta nel Capitolo 4 necessita di qualche leggero cambiamento; le dichiarazioni devono essere modificate, e le operazioni di confronto devono essere fatte chiamando la funzione `strcmp`. L'algo-ritmo rimane inalterato, e questo ci consente di sperare fondatamente che esso funzioni ancora.

```
/* qsort: ordina v[left] .... v[right] in ordine crescente */
void  qsort(char *v[], int left, int right)
{

    int i, last;
    void  swap(char *v[], int i, int j);

    if (left>=right)          /* se il vettore contiene meno di due */
        return;               /* elementi, non fa nulla */
    swap(v, left, (left+right)/2);
    last=left;
    for (i=left+1; i<=right; i++)
        if (strcmp(v[i], v[left])<0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
```

```
    qsort(v, last+1, right);
}
```

Per gli stessi motivi illustrati sopra, anche la funzione `swap` necessita di alcune banali modifiche:

```
/* swap: scambia v[i] con v[j] */
void  swap(char *v[], int i, int j)
{
    char *temp;

    temp=v[i];
    v[i]=v[j];
    v[j]=temp;
}
```

Poiché ogni singolo elemento di `v` (cioè di `lineptr`) è un puntatore a carattere, anche `temp` lo deve essere, in modo che un puntatore possa essere copiato nell'altro.

Esercizio 5.7 Riscrivete `readlines` in modo che memorizzi le linee di un vettore fornito dal `main`, invece di chiamare la funzione `alloc`. Di quanto risulta più veloce il programma?

5.7 Vettori Multidimensionali

Il C fornisce dei vettori multidimensionali rettangolari, anche se, nella pratica, essi vengono usati molto meno dei vettori di puntatori. In questa sezione, mostreremo alcune delle loro proprietà.

Consideriamo il problema della conversione della data, dal giorno del mese a quello dell'anno e viceversa. Per esempio, il primo Marzo è il sessantesimo giorno di un anno non bisestile, ed il sessantunesimo di uno bisestile. Definiamo due funzioni che effettuano le conversioni: `day_of_year` converte il giorno del mese nel giorno dell'anno, mentre `month_day` esegue la conversione opposta. Poiché quest'ultima funzione cal-cola due valori, gli argomenti mese e giorno saranno dei puntatori:

```
month_day(1988, 60, &m, &d)
```

assegna `2` ad `m` e `29` a `d` (29 Febbraio).

Entrambe queste funzioni hanno bisogno di alcune informazioni, che possono essere fornite in una tabella contenente il numero di giorni che compongono ogni singolo mese. Poiché il numero dei giorni di ogni mese dipende dal fatto che l'anno sia bisestile o meno, la cosa più semplice consiste nel suddividere i due casi in righe separate di un vettore bidimensionale, piuttosto che tenere traccia, durante tutta l'elaborazione, dei giorni che compongono il mese di Febbraio. Il vettore e le funzioni che effettuano le conversioni sono i seguenti:

```
static char daytab[2][13]={
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: calcola il giorno dell'anno */
int  day_of_year(int year, int month, int day)
{
    int i, leap;

    leap=year%4==0  &&  year%100!=0  ||  year%400==0;
    for (i=1; i<month; i++)
        day+=daytab[leap][i];
    return day;
}

/* month_day: calcola il giorno del mese */
void  month_day(int year, int yearday, int *pmonth, int *pday)
{
    int i, leap;
```

```
        leap=year%4==0  &&  year%100!=0  ||  year%400==0;
        for (i=1; yearday>daytab[leap][i]; i++)
            yearday-=daytab[leap][i];
        *pmonth=i;
        *pday=yearday;
    }
```

Ricordiamo che il valore aritmetico di un'espressione logica, come quello che viene assegnato a `leap`, è zero (falso) oppure uno (vero), ed in quanto tale può essere utilizzato come indice del vettore `daytab`.

Il vettore `daytab` dev'essere esterno a `day_of_year` e `month_day`, in modo che entrambi possano usarlo. L'abbiamo dichiarato di tipo `char`, per illustrare come i `char` possano essere utilizzati per memoriz-zare piccoli interi.

`daytab` è il primo vettore bidimensionale con il quale abbiamo a che fare. In C, un vettore bidimensionale è in realtà un vettore ad una dimensione, in cui ogni elemento è un vettore. Quindi, gli indici devono essere scritti come

```
daytab[i][j]            /* [riga][colonna] */
```

e non come

```
daytab[i, j]            /* SBAGLIATO */
```

A parte questa distinzione notazionale, un vettore bidimensionale può essere trattato in modo analogo a quello utilizzato negli altri linguaggi. Gli elementi vengono memorizzati per righe, quindi l'indice di destra, cioè la colonna, varia più velocemente, mano a mano che si accede agli elementi nell'ordine in cui sono memorizzati.

Un vettore viene inizializzato da una lista di inizializzatori racchiusi fra parentesi graffe; ogni riga di un vettore bidimensionale viene inizializzata con una lista corrispondente. Abbiamo iniziato il vettore `daytab` con una colonna di zeri, in modo che i numeri dei mesi possano variare da 1 a 12, anziché da 0 a 11. Poiché, in questa sede, non ci interessa risparmiare spazio, adottiamo questo accorgimento, che rende più chiaro l'impiego degli indici.

Se un vettore bidimensionale dev'essere passato ad una funzione, la dichiarazione del parametro nella funzione deve comprendere il numero delle colonne; il numero delle righe non è rilevante, perché ciò che viene passato, ancora una volta, è un puntatore ad un vettori di righe, nel quale ogni riga è un vettore di 13 interi. In questo caso particolare, esso è un puntatore ad oggetti che sono vettori di 13 interi. Quindi, se il vettore `daytab` dev'essere passato ad una funzione `f`, la dichiarazione di `f` deve avere la forma:

```
f(int daytab[2][13])  { .... }
```

Essa potrebbe anche essere scritta come

```
f(int daytab[][13])  { .... }
```

dato che il numero di righe è irrilevante. Oppure, ancora, si potrebbe scrivere

```
f(int (*daytab)[13])  { .... }
```

Quest'ultima dichiarazione afferma che il parametro è un puntatore ad un vettore di 13 interi. Le parentesi tonde sono necessarie perché quelle quadre `[]` hanno precedenza maggiore dell'operatore `*`. Senza parentesi, la dichiarazione

```
int *daytab[13]
```

crea un vettore di 13 puntatori ad interi. Più in generale, soltanto la prima dimensione (indice) di un vettore multidimensionale è libera; tutte le altre devono essere specificate.

La Sezione 5.12 approfondisce ulteriormente il problema di dichiarazioni complesse.

Esercizio 5.8 Nelle funzioni `day_of_year` e `month_day` non esiste alcun controllo sulle condizioni di errore. Colmate questa lacuna.

5.8 Inizializzazione di Vettori di Puntatori

Consideriamo il problema di scrivere una funzione `month_name(n)`, che ritorna il puntatore ad una stringa costante contenente il nome dell'`n`-esimo mese. Questa è una situazione alla quale ben si adatta l'impiego di un vettore `static` interno. `month_name` contiene un vettore privato di stringhe di caratteri e, quando viene chiamata, ritorna un puntatore all'elemento opportuno. Questa sezione illustra il modo in cui viene inizializzato il vettore dei nomi.

La sintassi è simile a quella delle inizializzazioni precedenti:

```
/* month_name: ritorna il nome dell'n-esimo mese */
char *month_name(int n)
{
    static char *name[]={
        "Mese sconosciuto",
        "Gennaio", "Febbraio", "Marzo",
        "Aprile", "Maggio", "Giugno",
        "Luglio", "Agosto", "Settembre",
        "Ottobre", "Novembre", "Dicembre"
    };

    return (n<1  ||  n>12)?name[0]:name[n];
}
```

La dichiarazione di `name`, che è un vettore di puntatori a carattere, è uguale a quella del vettore `lineptr`, nell'esempio relativo all'ordinamento. L'inizializzatore è una lista di stringhe di caratteri, ognuna delle quali viene assegnata alla corrispondente posizione all'interno del vettore. I caratteri dell'`i`-esima stringa vengo-no posti in una locazione qualsiasi, ed in `name[i]` viene memorizzato il puntatore a tale locazione. Poiché l'ampiezza del vettore `name` non è stata specificata, il compilatore conta gli inizializzatori e dimensiona il vettore di conseguenza.

5.9 Puntatori e Vettori Multidimensionali

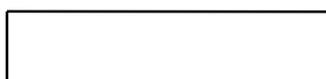
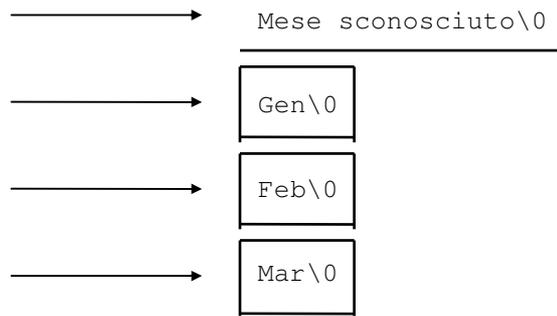
Spesso, per coloro che conoscono poco il C, la differenza fra vettori bidimensionali e vettori di puntatori, quali `name`, risulta confusa. Date le definizioni

```
int a[10][20];
int *b[10];
```

allora `a[3][4]` e `b[3][4]` sono entrambi dei riferimenti ad un singolo `int` sintatticamente corretti. Tutta-via, `a` è un vettore bidimensionale: per esso sono state riservate 200 locazioni di ampiezza pari a quella di un `int`, e l'elemento `a[`*riga*`][`*colonna*`]` viene individuato attraverso l'algoritmo 20**riga*+*colonna*. Per `b`, in-vece, la definizione alloca soltanto 10 puntatori, che non vengono inizializzati; l'inizializzazione dev'essere fatta esplicitamente, in modo statico oppure con del codice apposito. Assumendo che ogni elemento di `b` punti ad un vettore di 20 elementi, `b` stesso occuperà 200 allocazioni di ampiezza pari ad un `int`, più dieci celle per i puntatori. L'importante vantaggio offerto da un vettori di puntatori consiste nel fatto che esso consente di avere righe di lunghezza variabile. In altre parole, non è detto che ogni elemento di `b` punti ad un vettore di venti elementi; qualcuno può puntare a due elementi consecutivi, qualcun altro a cinquanta e qualcun altro ancora a zero.

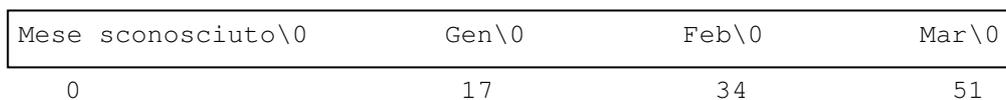
Anche se, finora, abbiamo parlato di interi, notiamo che l'uso più frequente dei vettori di puntatori consiste nel memorizzare stringhe di diversa lunghezza, come abbiamo fatto nella funzione `month_name`. Confron-tate la dichiarazione e lo schema seguenti, che descrivono un vettore di puntatori:

```
char *name[]={"Mese sconosciuto", "Gen", "Feb", "Mar"};

name:
```

con quelli relativi ad un vettore bidimensionale:

```
char aname[][17]={"Mese sconosciuto", "Gen", "Feb", "Mar"};
```

name:

Mese sconosciuto\0	Gen\0	Feb\0	Mar\0
0	17	34	51

Esercizio 5.9 Riscrivete la routine `day_of_year` e `month_day` usando puntatori invece che indici.

5.10 Argomenti alle Linee di Comando

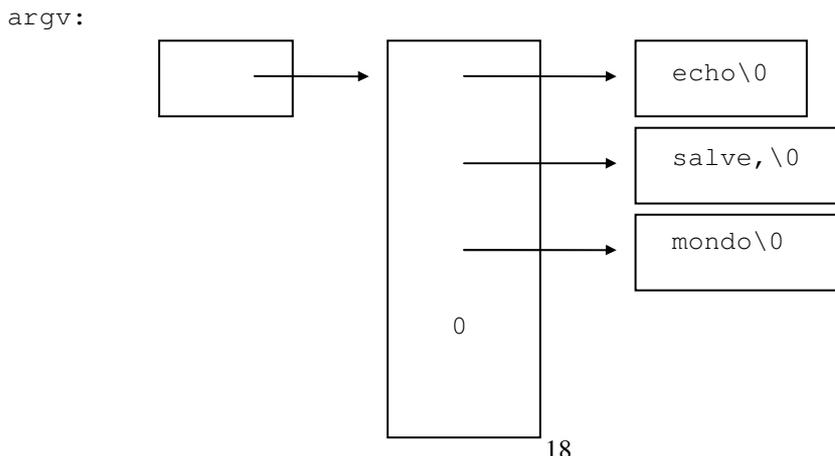
Negli ambienti che supportano il C, esiste un modo per passare argomenti o parametri in linea ad un programma quando questo viene eseguito. Eseguire un programma equivale a chiamare la funzione `main` definita al suo interno, alla quale vengono sempre passati due argomenti. Il primo (convenzionalmente chiama-to `argc`, da "argument count") è il numero degli argomenti in linea passati al programma; il secondo (`argv`, da "argument vector") è un puntatore ad un vettore di stringhe di caratteri che contengono gli argomenti, uno per stringa. Normalmente, per manipolare queste stringhe di caratteri noi utilizziamo più livelli di punta-tori.
L'esempio più semplice è dato dal programma `echo`, che stampa i suoi argomenti su una singola linea, separandoli con degli spazi. Cioè il comando

```
echo salve, mondo
```

stampa l'output

```
salve, mondo
```

Per convenzione, `argv[0]` contiene il nome con il quale il programma è stato invocato, quindi `argc` vale sempre almeno 1. Se `argc` è 1, dopo il nome del programma non esistono ulteriori argomenti. Nell'esempio precedente, `argc` era 3, e `argv[0]`, `argv[1]` e `argv[2]` erano, rispettivamente, `"echo"`, `"salve,"` e `"mondo"`. Il primo argomento opzionale è `argv[1]` mentre l'ultimo è `argv[argc-1]`; infine, lo standard richiede che `argv[argc]` sia un puntatore nullo.

La prima versione di `echo` tratta `argv` come se fosse un vettore di puntatori a caratteri:

```
#include  <stdio.h>

/* stampa gli argomenti alla linea di comando; prima versione */
main(int argc, char *argv[])
{
    int i;

    for (i=1; i<argc; i++)
        printf("%s%s", argv[i], (i<argc-1)?" ":"");
    printf("\n");
    return 0;
}
```

Poiché `argv` è un puntatore ad un vettore di puntatori, invece dell'indice noi possiamo manipolare il puntatore stesso. La prossima variante si basa sull'incremento di `argv`, che è un puntatore a dei puntatori a caratteri, e sul decremento di `argc`:

```
#include  <stdio.h>

/* stampa gli argomenti alla linea di comando; seconda versione */
main(int argc, char *argv[])
{
    while(--argc>0)
        printf("%s%s", *++argv, (argc>1)?" ":"");
    printf("\n");
    return 0;
}
```

Poiché `argv` è un puntatore all'inizio del vettore delle stringhe, incrementarlo di 1 (`++argv`) significa farlo puntare all'argomento che, inizialmente, avevamo indicato con `argv[1]`, anziché ad `argv[0]`. Ogni incremento successivo sposta il puntatore sull'argomento seguente; `*argv` diventa perciò il puntatore a tale argomento. Nello stesso tempo, `argc` viene decrementato; quando esso diventa zero gli argomenti da stampare sono esauriti.

Un'altra soluzione possibile consisteva nello scrivere l'istruzione `printf` nella forma seguente:

```
printf((argc>1)?"%s":"%s", *++argv);
```

Questa forma evidenza che l'argomento che descrive il formato, in una `printf`, può anche essere una espressione.

Come secondo esempio, apportiamo dei miglioramenti al programma di ricerca di un pattern illustrato nella Sezione 4.1. Se ben ricordate, la parte di ricerca del pattern era stata inserita molto all'interno del programma, e questa era una soluzione poco efficiente. Seguendo la linea del programma UNIX `grep`, modifichiamo il nostro programma in modo che il pattern da ricercare venga fornito come primo argomento della linea di comando.

```
#include  <stdio.h>
#include  <string.h>
#define  MAXLINE  1000

int  getline(char *line, int max);

/* find: stampa le linee che contengono il pattern fornito
   come primo argomento */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found=0;

    if (argc!=2)
        printf("Utilizzo: find pattern\n ");
    else
```

```
        while (getline(line, MAXLINE)>0)
            if (strstr(line, argv[])!=NULL)
            {
                printf("%s", line);
                found++;
            }
    return found;
}
```

La funzione `strstr(s, t)`, presente nella libreria standard, fornisce un puntatore alla prima occorrenza della stringa `t` nella stringa `s`, oppure `NULL`, se `t` non ricorre in `s`. Questa funzione è dichiarata in `<string.h>`.

Questo modello, ora, può essere rielaborato per illustrare ulteriori utilizzazioni dei puntatori. Supponiamo di volere consentire la presenza di due argomenti opzionali. Uno di questi argomenti significa "stampa tutte le linee *ad eccezione* di quelle contenenti il pattern specificato", l'altro significa "anteponi ad ogni linea stampata il suo numero d'ordine".

Una convenzione comune a tutti i programmi C che lavorano su sistemi UNIX è che un argomento che inizia con un segno meno indica un flag o un parametro opzionale. Se decidiamo che `-x` (da "except") indica l'inversione, e che `-n` (da "number") indica la richiesta di numerazione delle linee, allora il comando

`find -x -n` *`pattern`*

stampa ogni linea che non contiene il pattern, preceduta dal suo numero d'ordine.

Gli argomenti opzionali dovrebbero poter essere forniti in un ordine qualsiasi, ed il resto del programma dovrebbe essere indipendente dal numero di argomenti presenti. Inoltre, per gli utenti è conveniente che le opzioni possano essere combinate, come in

`find -nx` *`pattern`*

Il programma è il seguente :

```
#include  <stdio.h>
#include  <string.h>
#define  MAXLINE  1000

int getline(char *line, int max);

/* find: stampa le linee che contengono il pattern
   fornito come argomento */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    long lineno=0;
    int c, except=0, number=0, found=0;

    while (--argc>0  &&  (*++argv)[0]=='-')
        while (c=*++argv[0])
            switch (c)  {
            case 'x':
                except=1;
                break;
            case 'n':
                number=1;
                break;
            default:
                printf("find: opzione %c illegale\n", c);
                argc=0;
                found=-1;
                break;
            }
    if (argc!=1)
        printf("Utilizzo: find -x -n pattern\n");
```

```
        else
            while (getline(line, MAXLINE)>0)
            {
                lineno++;
                if ((strstr(line, *argv)!=NULL)!=except)
                {
                    if (number)
                        printf("%ld:", lineno);
                    printf("%s", line);
                    found++;
                }
            }
        return found;
    }
```

Prima di ogni argomento opzionale, `argc` viene decrementato ed `argv` viene incrementato. Al termine del ciclo, se non si sono verificati errori, `argc` ci dice quanti argomenti restano non analizzati, ed `argv` punta al primo di essi. Quindi, `argc` dovrebbe essere 1 e `*argv` dovrebbe puntare al pattern. Notate che `*++argv` è un puntatore ad una stringa, della quale `(*++argv)[0]` è il primo carattere (un'altra forma sintatticamen-te corretta potrebbe essere `**++argv`). Poiché le parentesi quadre `[]` hanno precedenza superiore a `++` e ad `*`, le parentesi tonde sono necessarie; senza di esse, l'espressione verrebbe considerata come `*++(argv[0])`. In realtà, questa è la forma che abbiamo utilizzato nel ciclo più interno, nel quale ci proponevamo di scandire una stringa specifica. In questo ciclo, l'espressione `*++argv[0]` incrementa il puntatore `argv[0]`!

Espressioni con puntatori più complesse di quelle mostrare in questo esempio sono raramente necessarie; in ogni caso, ove sia indispensabile utilizzarle, è meglio che esse vengano spezzate in due o tre fasi, in modo da risultare più intuitive.

Esercizio 5.10 Scrivete il programma `expr`, che valuta un'espressione in notazione Polacca inversa leggendola da una linea di comando, nella quale ogni operatore ed ogni operando costituiscono un argomento separato. Per esempio,

```
expr 2 3 4 + *
```

valuta 2*(3+4).

Esercizio 5.11 Modificate i programmi `entab` e `detab` (scritti per esercizio nel Capitolo 1) in modo che accettino come argomento una lista di tab stop. Se non ci sono argomenti, utilizzate i tab stop di default.

Esercizio 5.12 Estendete `entab` e `detab` in modo che accettino la notazione abbreviata

entab -*m* +*n*

che pone un tab stop ogni *n* colonne, a partire dalla colonna *m*. Scegliete dei valori di default opportuni (per l'utente).

Esercizio 5.13 Scrivete il programma `tail`, che stampa le ultime *n* linee del suo input. Per default, *n* vale 10, ma questo valore può essere modificato con un argomento opzionale, in modo che

tail -*n*

stampi le ultime *n* linee. Il programma dovrebbe comportarsi sensatamente indipendentemente dal fatto che il suo input o il valore di *n* siano ragionevoli. Scrivete il programma in modo che utilizzi al meglio la memo-ria; le linee dovrebbero essere memorizzate non in un vettore bidimensionale di ampiezza prefissata, bensì in modo analogo a quanto avveniva nel programma di ordinamento della Sezione 5.6.

5.11 Puntatori a Funzioni

In C, una funzione non è, di per se stessa, una variabile; tuttavia, è possibile dichiarare dei puntatori alle funzioni, e tali puntatori possono essere assegnati, inseriti in vettori, passati ad altre funzioni, restituiti e co-sì via. Illustreremo queste possibilità modificando la procedura di ordinamento scritta in questo capitolo in modo

che, se l'opzione -n è presente, le linee vengano ordinato in base al loro valore, piuttosto che lessicograficamente.

Una procedura di ordinamento, in genere, si compone di tre fasi: un confronto per determinare l'ordinamen-to di una qualsiasi coppia di oggetti, un'eventuale scambio che ne inverte l'ordine ed un algoritmo di ordinamento che ripete i confronti e le inversioni fino all'ottenimento della sequenza ordinata. L'algoritmo è indipendente dalle operazioni di confronto e di scambio quindi, passandogli differenti funzioni di confronto e di inversione, siamo in grado di ordinare gli elementi secondo criteri diversi.

Il confronto lessicografico fra due linee viene effettuato, ancora una volta, dalla funzione strcmp; ora, però, abbiamo bisogno anche di una funzione numcmp, che confronta due linee sulla base del loro valore numerico, e restituisce lo stesso tipo di informazioni restituite da strcmp. Queste funzioni sono dichiarate dopo il main, ed alla funzione qsort viene passato il puntatore ad una di esse. Nel nostro esempio abbiamo trascurato il controllo degli errori sul passaggio degli argomenti, in modo da poterci concentrare sugli obiettivi principali.

```
#include  <stdio.h>
#include  <string.h>

#define  MAXLINES  5000        /* numero massimo di linee */
char  *lineptr[MAXLINES];      /* puntatore alle linee di testo */

int  readlines(char *lineptr[], int nlines);
void  writelines(char *lineptr, int nlines);
void  qsort(void *lineptr, int left, int right,
            int (*comp)(void *, void *));
int numcmp(char *, char *);

/* ordina le linee di input */
main(int argc, char *argv[])
{
    int nlines;                /* numero di linee di input lette */
    int numeric=0;             /* 1 in caso di ordinamento numerico */

    if (argc>1  &&  strcmp(argv[1], “-n”)==0)
        numeric=1;
    if ((nlines=readlines(lineptr, MAXLINES))>=0)
    {
        qsort((void **) lineptr, 0 nlines-1,
        (int (*)(void *, void *))(numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    }
    else {
        printf(“Errore: troppe linee di input da ordinare\n”);
        return 1;
    }
}
```

Nella chiamata di qsort, strcmp e numcmp sono degli indirizzi di funzione. Poiché esse sono state dichiarate come funzioni, l'operatore & non è necessario, esattamente come nel caso di nomi di vettori.

Abbiamo riscritto qsort in modo che sia in grado di trattare qualsiasi tipo di dati, e non soltanto le stringhe di caratteri. Come è indicato dal prototipo della funzione, qsort si aspetta, come parametri, un vettore di puntatori, due interi ed una funzione con due puntatori come argomenti. Per questi ultimi, abbiamo usato il tipo riservato ai puntatori generici, void *. Qualsiasi puntatore può essere trasformato in void * e poi riportato al suo tipo originale senza alcuna perdita di informazione; questo ci consente di chiamare qsort forzando il tipo dei suoi argomenti a void *.

La complessa operazione di cast sugli argomenti della funzione riguarda gli argomenti della funzione di confronto. Questo tipo di operazioni, pur non avendo alcun effetto sulla rappresentazione reale, assicura al compilatore che la situazione è coerente.

```
/* qsort: ordina v[left] .... v[right] in ordine crescente */
```

```
void  qsort(void *v[], int left, int right,
            int (*comp)(void *, void *));
{
    int i, last;
    void  swap(void *v[], int, int);
    if (left>=right)          /* se il vettore contiene meno di */
        return;               /* due elementi, non fa nulla */
    swap(v, left, (left+right)/2);
    last=left;
    for (i=left+1; i<=right; i++)
        if ((*comp)(v[i], v[left])<0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
}
```

Le dichiarazioni meritano di essere analizzate con molta attenzione. Il quarto parametro di `qsort` è:

```
int (*comp)(void *, void *);
```

ed afferma che `comp` è un puntatore ad una funzione che ha due argomenti di tipo `void *` e che ritorna un `int`.

L'uso di `comp` nella linea

```
if ((*comp)(v[i], v[left])<0)
```

è consistente con la dichiarazione: `comp` è un puntatore ad una funzione, `*comp` è la funzione, e

```
(*comp)(v[i], v[left])
```

è la sua chiamata. Le parentesi sono necessarie, affinché le componenti vengano associate correttamente; senza di esse,

```
int *comp(void*, void *)            /* SBAGLIATO */
```

afferma che `comp` è una funzione che ritorna un puntatore ad un `int`, il che è molto diverso da quanto ci eravamo proposti.

Abbiamo già visto la funzione `strcmp`, che confronta due stringhe. Presentiamo ora `numcmp`, che confronta due stringhe sulla base del loro valore numerico, calcolato tramite la funzione `atof`:

```
#include  <stdlib.h>

/* numcmp: confronta numericamente s1 e s2 */
int  numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1=atof(s1);
    v2=atof(s2);
    if (v1<v2)
        return -1;
    else if (v1>v2)
        return 1;
    else
        return 0;
}
```

La funzione `swap`, che scambia fra loro due puntatori, è uguale a quella già presentata in questo capitolo, eccezione fatta per le dichiarazioni, che diventano `void *`:

```
void  swap(void *v[], int i, int j)
{
```

```
        void *temp;

        temp=v[i];
        v[i]=v[j];
        v[j]=temp;
    }
```

Al programma di ordinamento possono essere aggiunte molte opzioni, alcune delle quali vengono proposte negli esercizi seguenti.

Esercizio 5.14 Modificate il programma di ordinamento in modo che gestisca un'opzione `-r`, che indica la richiesta di ordinamento decrescente. Assicuratevi che `-r` operi correttamente anche in presenza dell'ordi-ne `-n`.

Esercizio 5.15 Aggiungete l'opzione `-f`, che annulla la distinzione fra lettere maiuscole e minuscole, in modo che, per esempio, `a` ed `A` vengano considerate uguali durante la procedura di ordinamento.

Esercizio 5.16 Aggiungete l'opzione `-d`, che effettua i confronti soltanto sulle lettere, sui numeri e sugli spazi. Assicuratevi che `-d` operi correttamente anche in presenza dell'opzione `-f`.

Esercizio 5.17 Aggiungete la possibilità di gestire dei campi, in modo che l'ordinamento possa essere fatto su campi all'interno delle linee, dove ogni campo viene ordinato in base ad un insieme autonomo di opzioni (l'indice di questo libro è ordinato con le opzioni `-df` per le categorie, e con `-n` per i numeri di pagina).

5.12 Dichiarazioni Complesse

Talvolta il C risulta molto vincolato dalla sintassi delle sue dichiarazioni, specialmente quelle relative ai puntatori alle funzioni. La sintassi non è altro che il tentativo di facilitare le dichiarazioni ed il loro uso; essa si adatta molto bene a casi relativamente semplici, ma può risultare confusa nei casi più complessi, perché le dichiarazioni non possono essere lette da sinistra a destra, e perché le parentesi vengono usate pesantemente. La differenza fra

```
/* f: funzione che ritorna un puntatore ad un int */
int *f();
```

e

```
/* pf: puntatore ad una funzione che ritorna un int */
int (*pf)();
```

esemplifica il problema: `*` è un operatore prefisso ed ha una precedenza inferiore a `()`, quindi, per forzare l'associazione corretta, le parentesi sono indispensabili.

Sebbene, nella pratica, le dichiarazioni molto complesse siano rare, è importante essere in grado di comprenderle e, se necessario, di crearle. Un buon metodo per sintetizzare le dichiarazioni consiste nel raggrupparle in piccole sezioni, usando l'istruzione `typedef`, che verrà discussa nella Sezione 6.7.

In alternativa, in questa sezione presenteremo una coppia di programmi che convertono codice C in descrizioni verbali e viceversa. Le descrizioni verbali si leggono da sinistra a destra.

Il primo programma, `dcl`, è il più complesso. Esso converte una dichiarazione C in una descrizione verbale, come per esempio:

```
char **argv
    argv: puntatore a dei puntatori a caratteri
int (*daytab)[13]
    daytab: puntatore ad un vettore di 13 int
int *daytab[13]
    daytab: vettore di 13 puntatori ad int
void  *comp()
    comp: funzione che ritorna un puntatore a void
void  (*comp)()
    comp: puntatore ad una funzione che ritorna un void
```

```
char  (*(*x())[])()
    x: funzione che ritorna un puntatore ad un vettore di
    puntatori a funzioni che ritornano un char
char  (*(*x[3])())[5]
    x: vettore di 3 puntatori a funzioni che ritornano un
    puntatore ad un vettore di 5 char
```

`dcl` si basa sulla grammatica che specifica un dichiaratore, espressa in modo preciso nell'Appendice A, Sezione 8.5; la forma semplificata di tale grammatica è la seguente:

```
dcl:                    uno o più * opzionali  direct-dcl
direct-dcl:             nome
                        (dcl)
                        direct-dcl()
                        direct-dcl[ampiezza opzionale]
```

Verbalmente, questo equivale a dire che *dcl* è un *direct-dcl*, che può anche essere preceduto da uno o più `*`. Un *direct-dcl* è un nome, oppure un *dcl* fra parentesi tonde, oppure un *direct-dcl* seguito da una coppia di parentesi tonde, oppure un *direct-dcl* seguito da parentesi quadre contenenti un'ampiezza opzionale.
Questa grammatica può essere utilizzata per analizzare le dichiarazioni. Per esempio, consideriamo il dichiaratore:

```
(*pfa[])()
```

`pfa` viene identificato come un *nome*, cioè come un *direct-dcl*. Quindi anche `pfa[]` è un *direct-dcl*. Da questo segue che `*pfa[]` viene riconosciuto come un *dcl*, quindi `(*pfa[])` è un *direct-dcl*.

Allora, `(*pfa[])()` è un *direct-dcl*, quindi un *dcl*. Questa procedura di analisi può essere illustrata anche tramite un algoritmo di parsing come il seguente (nel quale il termine *direct-dcl* è stato abbreviato in *dir-dcl*):

Il cuore del programma `dcl` è costituito dalla coppia di funzioni, `dcl` e `dirdcl`, che analizzano una dichiarazione sulla base della grammatica data. Poiché la grammatica è definita ricorsivamente, le funzioni si richiamano ricorsivamente mano a mano che riconoscono segmenti di una dichiarazione; il programma viene chiamato analizzatore (parser) discendente ricorsivo.

```
/* dcl: analizza un dichiaratore */
void  dcl(void)
{
    int ns;

    for (ns=0; gettoken()=='*';)          /* conta gli * */
        ns++;
    dirdcl();
    while (ns-->0)
        strcat(out, " puntatore a");
}

/* dirdcl: analizza un dichiaratore diretto */
void  dirdcl(void)
```

```
{
    int type;

    if (tokentype=='(')                   /* ( dcl ) */
    {
        dcl();
        if (tokentype!=')')
            printf("Errore: manca una parentesi )\n");
    }
    else if (tokentype==NAME)             /* nome variabile */
        strcpy(name, token);
    else
        printf("Errore: atteso un nome o un (dcl)\n");
    while ((type=gettoken())==PARENS  ||  type==BRACKETS)
        if (type==PARENS)
            strcat(out, " funzione che ritorna");
        else
        {
            strcat(out, " vettore");
            strcat(out, token);
            strcat(out, " di");
        }
}
```

Poiché i nostri programmi vogliono essere soltanto dimostrativi, la funzione `dcl` è soggetta a restrizioni piuttosto significative. Essa può gestire soltanto tipi di dati molto semplici, come i `char` o gli `int`. Essa non gestisce i tipi degli argomenti delle funzioni, o i qualificatori come `const`. Gli spazi in sovrannumero la confondono. Poiché non gestisce molto bene neppure gli errori, anche le dichiarazioni scorrette la confondono. Tutte le modifiche necessarie ad eliminare queste limitazioni vengono lasciate come esercizio.

Di seguito presentiamo le variabili globali e la routine principale:

```
#include  <stdio.h>
#include  <string.h>
#include  <ctype.h>

#define  MAXTOKEN  100

enum {NAME, PARENS, BRACKETS};

void  dcl(void);
void  dirdcl(void);
int  gettoken(void);
int  tohentype;               /* tipo dell'ultimo token */
char  token[MAXTOKEN];        /* ultimo token */
char  name[MAXTOKEN];         /* nome identificatore */
char  datatype[MAXTOKEN];     /* tipo di dati = char, int .... */
char  out[1000];              /* stringa di output */

main()            /* converte le dichiarazioni in descrizioni verbali */
{
    while (gettoken()!=EOF)
    {
        strcpy(datatype, token);    /* il primo token della linea
                                       è il tipo dei dati */
        out[0]='0';
        dcl();                      /* analizza il resto della linea */
        if (tokentype!='\n')
            printf("Errore di sintassi\n");
        printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}
```

La funzione `gettoken` tralascia gli spazi ed i caratteri di tabulazione, e trova il token successivo nell'input; un "token" è un nome, una coppia di parentesi tonde, una coppia di parentesi quadre che possono anche racchiudere un numero, oppure un qualsiasi altro singolo carattere.

```
int  gettoken(void)          /* ritorna il token successivo */
{
    int c, getch(void);
    void  ungetch(int);
    char *p=token;

    while ((c=getch())==' '  ||  c=='\t')
        ;
    if (c=='(')
    {
        if ((c==getch())=='(')
        {
            strcpy(token, "()");
            return tokentype=PARENS;
        }
        else
        {
            ungetch(c);
            return tokentype='(';
        }
    }
    else if (c=='(')
    {
        for (*p++=c; (*p++=getch())!=']';)
            ;
        *p='\0';
        return tokentype=BRACKETS;
    }
    else if (isalpha(c))
    {
        for (*p++=c; isalnum(c=getch());)
            *p++=c;
        *p='\0';
        ungetch(c);
        return tokentype=NAME;
    }
    else
        return tokentype=c;
}
```

`getch` e `ungetch` sono state discusse nel Capitolo 4.

Il percorso inverso è più semplice, specialmente se non ci poniamo il problema di evidenziare la generazione di parentesi ridondanti. Il programma `undcl` converte una descrizione verbale come "`x` è una funzione che ritorna un puntatore ad un vettore di puntatori a funzioni che ritornano un `char`", che esprimeremo come

```
x  ()  *  []  *  ()  char
```

nella dichiarazione

```
char (*(*x())[])()
```

La sintassi abbreviata dell'input ci consente di riutilizzare la funzione `gettoken`. Inoltre, `undcl` usa anche le stesse variabili esterne impiegate da `dcl`.

```
/* undcl: converte una descrizione verbale in una dichiarazione */
main()
{
    int type;
    char temp[MAXTOKEN];

    while(gettoken()!=EOF)
```

```
        {
            strcpy(out, token);
            while ((type=gettoken())!='\n')
                if (type==PARENS ||  type==BRACKETS)
                    strcat(out, token);
                else
                {
                    sprintf(temp, "(*%s)", out);
                    strcpy(out, temp);
                }
                else if (type==NAME)
                {
                    sprintf(temp, "%s %s", token, out);
                    strcpy(out, temp);
                }
                else
                    printf("Input illegale a %s\n", token);
            printf("%s\n", out);
        }
        return 0;
    }
```

Esercizio 5.18 Fate in modo che `dcl` gestisca gli errori sull'input.

Esercizio 5.19 Modificate `undcl` in modo che non aggiunga parentesi ridondanti alle dichiarazioni.

Esercizio 5.20 Estendete `dcl` in modo che gestisca dichiarazioni contenenti i tipi degli argomenti di funzione, i qualificatori come `const` e così via.

CAPITOLO 6

STRUTTURE

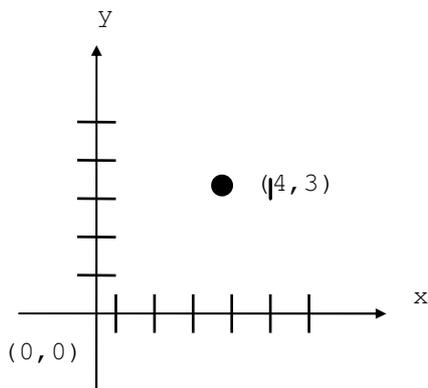
Una struttura è una collezione contenente una o più variabili, di uno o più tipi, raggruppate da un nome comune per motivi di maneggevolezza (in alcuni linguaggi, come il Pascal, le strutture vengono chiamate "record"). Soprattutto in programmi di dimensioni notevoli, le strutture aiutano ad organizzare dati comples-si, in quanto consentono di trattare come un unico oggetto un insieme di variabili correlate.

Un esempio classico di struttura è quello dello stipendio: un impiegato viene descritto da un insieme di attributi, quali il nome, l'indirizzo, il numero della tessera sanitaria, lo stipendio e così via. A loro volta, alcuni di questi attributi potrebbero essere delle strutture: ogni nome ha diverse componenti, così come avviene per ogni indirizzo o stipendio. Un altro esempio, più tipico del C, proviene dalla grafica: un punto è una coppia di coordinate, un rettangolo è una coppia di punti e così via.

La principale variazione introdotta dall'ANSI C riguarda l'assegnamento delle strutture, che possono essere copiate ed assegnate, passate alle funzioni e da queste restituite.

Molti compilatori forniscono già da alcuni anni queste funzionalità, le quali tuttavia vengono precisate soltanto ora. Adesso, anche le strutture automatiche ed i vettori possono essere inizializzati.

6.1 Fondamenti sulle Strutture

Creiamo alcune strutture utili per la grafica. L'oggetto base è un punto, che noi assumiamo abbia due coordinate intere, *x* ed *y*.

Queste due componenti possono essere collocate in una struttura, dichiarata nel seguente modo:

```
struct point  {
    int x;
    int y;
};
```

La parola chiave `struct` introduce una dichiarazione di struttura, che è una lista di dichiarazioni racchiuse fra parentesi graffe. Un nome opzionale (`point` nel nostro esempio), chiamato identificatore o *tag della struttura*, può seguire la parola `struct`. Il tag identifica questo tipo di struttura, e può essere utilizzato co-me abbreviazione per la parte di dichiarazione compresa fra le parentesi.

Le variabili specificate nella struttura sono dette *membri*. Il membro di una struttura, un tag ed una variabile ordinaria (che, cioè, non appartiene ad alcuna struttura) possono avere lo stesso nome senza che questo crei dei conflitti, poiché il contesto consente sempre di distinguerli. Inoltre, gli stessi nomi dei membri possono ricorrere in strutture diverse, anche se, per chiarezza, è sempre meglio usare nomi uguali soltanto per oggetti strettamente correlati.

Una dichiarazione `struct` definisce un tipo. La parentesi graffa di chiusura che chiude la lista dei membri può essere seguita da una lista di variabili, analogamente a quanto avviene per i tipi fondamentali. Cioè,

```
struct  { .... }  x, y, z;
```

è sintatticamente analogo a

```
int x, y, z;
```

nel senso che ognuna di queste due istruzioni dichiara tre variabili (`x`, `y` e `z`) del tipo voluto, e riserva lo spazio necessario per esse.

Una dichiarazione di struttura non seguita da una lista di variabili non riserva alcun'area di memoria; essa descrive soltanto l'aspetto della struttura. Se la dichiarazione ha però un tag, esso può essere utilizzato in un secondo tempo nelle definizioni che istanziano la struttura. Per esempio, data la precedente dichiarazio-ne di `point`,

```
struct point  pt;
```

definisce una variabile `pt` come una struttura di tipo `struct point`. Una struttura può venire inizializzata facendo seguire la sua definizione da una lista di inizializzatori, ognuno dei quali è un'espressione costante:

```
struct point  maxpt={320, 200};
```

Una struttura automatica può venire inizializzata anche tramite un assegnamento, oppure una chiamata ad una funzione che ritorna una struttura del tipo corretto.

In un'espressione, un membro di una particolare struttura viene individuato attraverso un costrutto del tipo

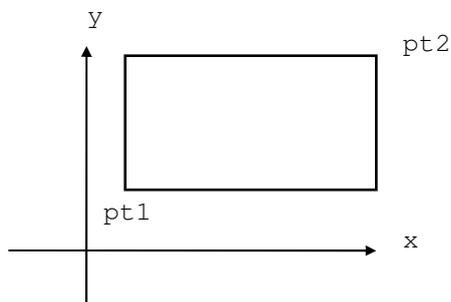
nome-struttura.membro

L'operatore di membro di una struttura "`.`" connette il nome della struttura con quello del membro. Per esempio, per stampare le coordinate del punto `pt` possiamo scrivere:

```
printf("%d,%d", pt.x, pt.y);
```

mentre, per calcolare la distanza di `pt` dall'origine (0,0) scriviamo:

```
double dist, sqrt(double);
dist=sqrt((double)pt.x*pt.x+(double)pt.y*pt.y);
```

Le strutture possono essere nidificate l'una nell'altra. Una rappresentazione di un rettangolo è una coppia di punti che denotano i due angoli diagonalmente opposti:

```
struct rect  {
    struct point pt1;
    struct point pt2;
};
```

La struttura `rect` contiene due strutture `point`. Se dichiariamo `screen` come

```
struct rect  screen;
```

allora

```
screen.pt1.x
```

si riferisce alla coordinata *x* del membro `pt1` di `screen`.

6.2 Strutture e Funzioni

Una struttura può essere soltanto copiata, assegnata come un unico oggetto, indirizzato tramite l'operatore `&`, oppure manipolata tramite l'accesso ai suoi membri. La copia e l'assegnamento comprendono anche il passaggio di argomenti alle funzioni e la restituzione di valori dalle funzioni. Le strutture non possono esse-re confrontate. Una struttura può essere inizializzata con una lista di valori costanti, uno per membro; una struttura automatica, poi, può essere inizializzata anche tramite un assegnamento.

Approfondiamo lo studio delle strutture scrivendo alcune funzioni che manipolano punti e rettangoli. Gli approcci possibili sono almeno tre: passare separatamente le componenti, passare un'intera struttura o passare un puntatore ad essa. Ognuna di queste soluzioni possiede vantaggi e svantaggi.

La prima funzione, `makepoint`, riceve in input due interi e ritorna una struttura `point`:

```
/* makepoint: crea un punto a partire da x e y */
struct point  makepoint(int x, int y)
{
    struct point  temp;

    temp.x=x;
    temp.y=y;
    return temp;
}
```

Notate che non esiste conflitto alcuno fra i nomi degli argomenti e quelli dei membri della struttura; al contrario, l'omonimia evidenzia la relazione fra di essi.

Ora `makepoint` può essere utilizzata per inizializzare dinamicamente una qualsiasi struttura, oppure per passare delle strutture come argomenti ad una funzione.

```
struct rect  screen;
struct point  middle;
struct point  makepoint(int, int);

screen.pt1=makepoint(0, 0);
screen.pt2=makepoint(XMAX, YMAX);
middle=makepoint((screen.pt1.x+screen.pt2.x)/2,
                 (screen.pt1.y+screen.pt2.y)/2);
```

Il passo successivo consiste nella stesura di funzioni che svolgono operazioni aritmetiche sui punti. Per esempio,

```
/* addpoint: somma due punti */
struct point  addpoint(struct point  pt1, struct point  p2);
{
    p1.x+=p2.x;
    p1.y+=p2.y;
    return p1;
}
```

In questa funzione, sia gli argomenti che il valore ritornano sono delle strutture. Invece di usare una variabi-le temporanea apposita, abbiamo incrementato le componenti di `p1`, in modo da evidenziare il fatto che i parametri strutturati vengono passati per valore come qualsiasi altro.

Come ulteriore esempio, scriviamo la funzione `ptinrec`, che controlla se un punto si trova all'interno di un rettangolo dato; la convenzione adottata è che un rettangolo includa i suoi lati sinistro ed inferiore, ed escluda quelli superiore e destro.

```
/* ptinrec: ritorna 1 se p è in r, 0 altrimenti */
```

```
int  ptinrec(struct point  p, struct rect  r)
{
    return p.x>=r.pt1.x  &&  p.x<r.pt2.x  &&
           p.y>=r.pt1.y  &&  p.y<r.pt2.y;
}
```

Questa funzione assume che un rettangolo sia rappresentato in forma canonica, cioè che le coordinate di `pt1` siano inferiori a quelle di `pt2`. La funzione seguente restituisce un rettangolo per il quale garantisce la forma canonica:

```
#define  min(a, b)  ((a)<(b)?(a):(b))
#define  max(a, b)  ((a)>(b)?(a):(b))

/* canonrect: rende canoniche le coordinate di un rettangolo */
struct rect  canonrect(struct rect  r)
{
    struct rect  temp;

    temp.pt1.x=min(r.pt1.x, r.pt2.x);
    temp.pt1.y=min(r.pt1.y, r.pt2.y);
    temp.pt2.x=max(r.pt1.x, r.pt2.x);
    temp.pt2.y=max(r.pt1.y, r.pt2.y);

    return temp;
}
```

Se ad una funzione dev'essere passata una struttura molto grande, in genere è più conveniente passare il puntatore alla struttura stessa. I puntatori alle strutture sono del tutto analoghi a quelli delle variabili ordinarie. La dichiarazione

```
struct point  *pp;
```

afferma che `pp` è un puntatore ad una struttura di tipo `struct point`. Se `pp` punta ad una struttura `point`, `*pp` è la struttura, mentre `(*pp).x` e `(*pp).y` sono i membri. Per usare `pp` potremmo scrivere, per esempio

```
struct point  origin, *pp;

pp=&origin;
printf("L'origine è (%d, %d)\n", (*pp).x, (*pp).y);
```

Le parentesi sono necessarie in `(*pp).x`, perché la precedenza dell'operatore di membro di una struttura, "`.`", è superiore a quella di `*`. L'espressione `*pp.x` significa perciò `*(pp.x)` che è scorretto, in quanto `x` non è un puntatore.

I puntatori alle strutture sono usati tanto spesso che, per brevità, si è deciso di non fornire una notazione alternativa. Se `p` è un puntatore ad una struttura, allora

`p->`*`membro-della-struttura`*

si riferisce al membro nominato(l'operatore `->` è un segno meno immediatamente seguito da un maggiore). Quindi, potremmo anche scrivere

```
printf("L'origine è (%d, %d)\n", pp->x, pp->y);
```

Sia `.` che `->` sono operatori associativi da sinistra a destra, pertanto se abbiamo

```
struct rect  r, *rp=r;
```

allora le seguenti quattro espressioni sono equivalenti:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Gli operatori di struttura `.` e `->`, insieme a `()` per le chiamate di funzione ed a `[]` per gli indici, sono collo-cati al vertice della scala delle precedenze e, di conseguenza, sono molto vincolanti. Per esempio, data la dichiarazione

```
struct {
    int len;
    char *str;
} *p;
```

allora

```
++p->len;
```

incrementa `len` e non `p`, perché la parentesizzazione sottintesa è `++(p->len)`. L'ordine di valutazione può venire alterato tramite le parentesi: `(++p)->len` incrementa `p` prima di accedere a `len`, mentre `(p++)->len` lo incrementa dopo (quest'ultima coppia di parentesi non è necessaria).

Analogamente, `*p->str` accede a ciò che è puntato da `str`; `*p->str++` incrementa `str` dopo aver acceduto al suo contenuto (come `*str++`); `(*p->str)++` incrementa l'oggetto puntato da `str`; infine, `*p++->str` incrementa `p` dopo aver acceduto all'oggetto puntato da `str`.

6.3 Vettori di Strutture

Vogliamo scrivere un programma che conti le occorrenze di ogni parola chiave del C. Abbiamo bisogno di un vettore di stringhe di caratteri che contenga i nomi, e di un vettore di interi che contenga i contatori. Una possibile soluzione consiste nell'impiego di due vettori paralleli, che potremmo chiamare `keyword` e `keycount`, come in

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

Tuttavia, il fattore stesso che i vettori siano paralleli suggerisce un'organizzazione diversa: un vettore di strutture. Ogni elemento che descrive una parola chiave è una coppia:

```
char *word;
int count;
```

ed ecco quindi definito un vettore di coppie. La dichiarazione di struttura

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

dichiara un tipo di struttura, `key`, definisce un vettore `keytab` di strutture di questo tipo, e riserva memoria per tali strutture. Ogni elemento del vettore è una struttura. Avremmo anche potuto scrivere:

```
struct key {
    char *word;
    int count;
};

struct key  keytab[NKEYS];
```

Poiché la struttura `keytab` contiene un insieme costante di stringhe, è meglio dichiararla come variabile esterna ed inizializzarla una volta per tutte al momento della sua definizione. L'inizializzazione della struttu-ra è analoga a quelle precedenti: la definizione è seguita da una lista di inizializzatori racchiusi fra parente-si graffe:

```
struct key {
    char *word;
    int count;
} keytab[]={
```

```
    "auto", 0, "break", 0, "case", 0, "char",
    0, "const", 0, "continue", 0, "default", 0,
    /* .... */ "unsigned", 0, "void", 0,
    "volatile", 0, "while", 0
}
```

Gli inizializzatori sono elencati in coppie che corrispondono ai membri della struttura. Sarebbe più preciso racchiudere gli inizializzatori di ogni riga fra parentesi graffe, come in

```
{ "auto", 0 },
{ "break", 0 },
{ "case", 0 },
....
```

ma, quando gli inizializzatori sono semplici variabili o stringhe di caratteri e sono tutti presenti, le parentesi interne non sono necessarie. Come al solito, il numero di elementi del vettore `keytab` verrà calcolato soltanto se le parentesi quadre `[]` saranno vuote e se verranno forniti tutti gli inizializzatori.

Il programma di conteggio delle parole chiave inizia con la definizione del vettore `keytab`. La routine principale legge l'input chiamando ripetutamente la funzione `getword`, che legge una parola per volta. Ogni parola viene cercata in `keytab`, utilizzando una versione del programma di ricerca binaria presentato nel Capitolo 3. La lista delle parole chiave, all'interno della tabella, dev'essere ordinata in ordine alfabetico crescente.

```
#include  <stdio.h>
#include  <ctype.h>
#include  <string.h>

#define  MAXWORD  100

int  getword(char *, int);
int  binsearch(char *, struct key *, int);

/* conta le parole chiave C */
main()
{
    int n;
    char word[MAXWORD];

    while (getword(word, MAXWORD)!=EOF)
        if (isalpha(word[0]))
            if ((n=binsearch(word, keytab, NKEYS))>=0)
                keytab[n].count++;
    for (n=0; n<NKEYS; n++)
        if (keytab[n].count>0)
            printf("%4d %s\n", keytab[n].count, keytab[n].word);
    return 0;
}

/* binsearch: trova word in tab[0] .... tab[n-1] */
int  binsearch(char *word, struct key  tab[], int n)
{
    int cond;
    int low, high, mid;

    low=0;
    high=n-1;
    while (low<=high)
    {
        mid=(low+high)/2;
        if ((cond=strcmp(word, tab[mid].word))<0)
            high=mid-1;
        else if (cond>0)
            low=mid+1;
        else
            return mid;
```

```
    }
    return -1;
}
```

Illustreremo la funzione `getword` in un secondo tempo; per ora, è sufficiente sapere che ogni chiamata a `getword` legge una parola, che viene copiata nel vettore passato come primo argomento a `getword` stessa.

La quantità `NKEYS` è il numero di parole chiave contenute in `keytab`. Sebbene sia possibile calcolarla a mano, è più semplice e più sicuro affidare questo compito alla macchina, specialmente se la lista è soggetta a cambiamenti. Una possibilità consiste nel terminare la lista degli inizializzatori con un puntatore nullo e, quindi, contare gli elementi di `keytab` scandendo la tabella fino al suo raggiungimento.

Questa soluzione è però più complessa del necessario, perché l'ampiezza del vettore è completamente determinata al momento della compilazione. La dimensione del vettore è data dalla dimensione di un elemento moltiplicata per il numero di elementi, quindi quest'ultimo è, a sua volta, dato da

dimensione di `keytab` / *dimensione di* `struct key`

Il C fornisce un operatore unario, chiamato `sizeof`, che può essere utilizzato per calcolare la dimensione di un oggetto. Le espressioni

`sizeof` *oggetto*

e

`sizeof (`*nome di un tipo*`)`

producono un intero uguale all'ampiezza, in byte, dell'oggetto o del tipo specificati (precisamente, `sizeof` produce un intero privo di segno il cui tipo, `size_t`. è definito nell'header `<stddef.h>`). Un oggetto può essere una variabile od un vettore, oppure ancora una struttura. Un nome di tipo può essere il nome di un tipo fondamentale, come `int` o `double`, oppure quello di un tipo derivato, come una struttura o un puntatore.

Nel nostro caso, il numero di parole chiave è dato dall'ampiezza del vettore divisa per la dimensione di un suo elemento. Questo calcolo viene utilizzato in un'istruzione `#define`, per definire il valore di `NKEYS`:

```
#define  NKEYS  (sizeof keytab / sizeof(struct key))
```

Un altro modo per effettuare questo calcolo è il seguente:

```
#define  NKEYS  (sizeof keytab / sizeof keytab[0])
```

Questa seconda forma presenta il vantaggio di non dover essere modificata nel caso in cui il nome del tipo venga cambiato.

L'operatore `sizeof` non può essere utilizzato in una riga che inizi con `#if`, perché il preprocessore non analizza i nomi dei tipi. Al contrario, l'espressione nella riga `#define` non viene valutata dal preprocessore e risulta quindi legale.

Occupiamoci ora della funzione `getword`. La versione che abbiamo scritto è più generale di quanto non sia necessario per questo programma, tuttavia non è complicata. `getword` legge la "prossima" parola dall'in-put, dove una parola può essere una stringa di lettere e cifre che inizia con una lettera, oppure un singolo carattere non bianco. Il valore della funzione è il primo carattere della parola, `EOF` in caso di fine dell'input, oppure il carattere stesso se non è alfabetico.

```
/* getword: legge la prossima parola o carattere dall'input */
int  getword(char *word, int lim)
{
    int c, getch(void);
    void  ungetch(int);
    char *w=word;
```

```
    while (isspace(c=getch()))
        ;
    if (c!=EOF)
        *w++=c;
    if (!isalpha(c))
    {
        *w='\0';
        return c;
    }
    for ( ; --lim>0; w++)
        if (!isalnum(*w=getch()))
        {
            ungetch(*w);
            break;
        }
    *w='\0';
    return word[0];
}
```

getword usa le funzioni getch e ungetch che abbiamo scritto nel Capitolo 4. Quando termina la lettura di un token alfanumerico, getword ha già letto un carattere più del dovuto. La chiamata a ungetch restitui-sce tale carattere all'input, in modo che venga preso in considerazione alla chiamata successiva di get-word. Altre funzioni usate da getword sono isspace per tralasciare gli spazi bianchi, isalpha per identi-ficare le lettere e isalnum per identificare lettere e cifre; tutte queste funzioni provengono dall'header stan-dard <ctype.h>.

Esercizio 6.1 La nostra versione di getword non gestisce correttamente gli underscore, le costanti di tipo stringa, i commenti e le linee di controllo del preprocessore. Scrivete una migliore versione di questa funzio-ne.

6.4 Puntatori a Strutture

Per illustrare alcune delle considerazioni legate ai vettori di strutture ed ai puntatori ad essi, riscriviamo il programma di conteggio delle parole chiave, usando però dei puntatori al posto degli indici dei vettori.

La dichiarazione esterna di keytab non subisce alcuna variazione, al contrario di quanto avviene per le funzioni main e binsearch.

```
#include  <stdio.h>
#include  <ctype.h>
#include  <string.h>
#define  MAXWORD  100

int  getword(char *, int);
struct key  *binsearch(char *, struct key *, int);

/* conta le parole chiave C; versione con puntatori */
main()
{
    char word[MAXWORD];
    struct key  *p;

    while (getword(word, MAXWORD)!=EOF)
        if (isalpha(word[0]))
            if ((p=binsearch(word, keytab, NKEYS))!=NULL)
                p->count++;
    for (p=keytab; p<keytab+NKEYS; p++)
        if (p->count>0)
            printf("%4d %s\n", p->count, p->word);
    return 0;
}

/* binsearch: trova word in tab[0] .... tab[n-1] */
struct key  *binsearch(char *word, struct key  *tab, int n)
{
```

```
    int cond;
    struct key  *low=&tab[0];
    struct key  *high=&tab[n];
    struct key  *mid;

    while (low<high)
    {
        mid=low+(high-low)/2;
        if ((cond=strcmp(word, mid->word))<0)
            high=mid;
        else if (cond>0)
            low=mid+1;
        else
            return mid;
    }
    return NULL;
}
```

Questo programma ha alcune caratteristiche degne di nota. Innanzitutto, la dichiarazione di `binsearch` deve indicare che questa funzione ritorna un puntatore ad un oggetto di tipo `struct key`, piuttosto che un intero; tale proprietà viene enunciata sia in `binsearch` che nel suo prototipo. Se `binsearch` trova la parola, ritorna il puntatore ad essa; se non la trova, ritorna `NULL`.

In secondo luogo, ora agli elementi di `keytab` si accede tramite puntatori. Questo comporta modifiche significative all'interno di `binsearch`.

Gli inizializzatori per `low` ed `high`, adesso, sono dei puntatori all'inizio ed alla fine di `keytab`.

Il calcolo dell'elemento centrale non può più essere semplicemente

```
mid=(low+high)/2;       /* SBAGLIATO */
```

perché la somma fra puntatori non è consentita. Tuttavia, la sottrazione è legale, quindi `high-low` è il numero degli elementi, e di conseguenza

```
mid=low+(high-low)/2;
```

fa in modo che `mid` punti all'elemento mediano fra `low` ed `high`.

La modifica basilare è, comunque, quella che impedisce all'algoritmo di generare un puntatore illegale o di accedere ad un elemento esterno al vettore. Il problema è dato dal fatto che sia `&tab[-1]` che `&tab[n]` cadono al di fuori dei limiti del vettore `tab`. Il primo dei due riferimenti è strettamente illegale, mentre il secondo, pur essendo sintatticamente corretto, è illegale in quanto accede ad un elemento esterno al vettore. Tuttavia, la definizione del linguaggio garantisce che le operazioni aritmetiche su puntatori che puntano al primo elemento oltre la fine di un vettore (cioè `&tab[n]`) lavorino correttamente.

Nel `main` abbiamo scritto

```
for (p=keytab; p<keytab+NKEYS; p++)
```

Se `p` è un puntatore ad una struttura, l'aritmetica su `p` tiene in considerazione la dimensione della struttura, quindi `p++` incrementa `p` di una quantità tale da farlo puntare alla struttura successiva, ed il test conclude il ciclo al momento opportuno.

Non possiamo comunque assumere che la dimensione di una struttura sia pari alla somma delle dimensioni dei suoi membri. A causa di particolari requisiti di allineamento su oggetti differenti, all'interno di una struttura possono esserci "buchi" privi di nome. Se, per esempio, un `char` occupa un byte ed un `int` ne occupa quattro, la struttura

```
struct {
    char c;
    int i;
};
```

potrebbe anche occupare otto byte invece di cinque. L'operatore `sizeof` ritorna il valore corretto.

Infine, facciamo un'osservazione legata al formato del programma: quando una funzione ritorna del tipo complesso, come il puntatore ad una struttura, come in

```
struct key  *binsearch(char *word, struct key  *tab, int n)
```

il nome della funzione può risultare poco visibile. Per questo motivo, spesso viene utilizzato lo stile se-guente

```
struct key  *
binsearch(char *word, struct key  *tab, int n)
```

La scelta dello stile è personale; scegliete la forma che preferite ed attenetevi ad essa.

6.5 Strutture Ricorsive

Supponiamo di volere gestire il programma più generale di contare le occorrenze di *tutte* le parole presenti in un particolare input. Poiché la lista delle parole non è nota a priori, non siamo in grado di ordinarla e di utilizzare la ricerca binaria.

Non possiamo neppure effettuare una ricerca lineare su ogni parola in input, per vedere se essa è già stata letta in precedenza: il programma risulterebbe troppo lungo (in particolare, il suo tempo d'esecuzione cre-scerebbe quadraticamente rispetto al crescere delle parole in input). Come possiamo organizzare i dati, in modo da gestire con efficienza una lista di parole arbitrarie?

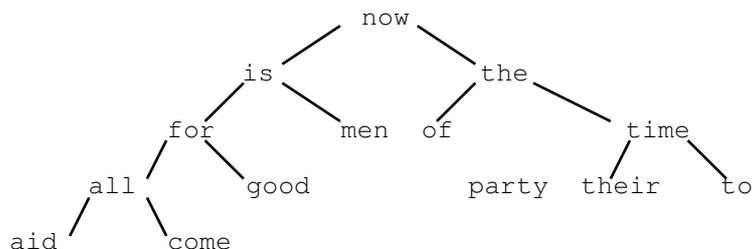
Una soluzione consiste nel tenere sempre ordinato l'insieme delle parole lette fino ad un certo istante, collo-cando nella posizione corretta ogni parola in arrivo. Ciò non può essere fatto spostando le parole all'interno di un vettore lineare, perché anche questo procedimento risulterebbe eccessivamente lungo. Useremo inve-ce una struttura dati detta *albero binario*.

L'albero contiene un "nodo" per ogni parola diversa; ogni nodo contiene:

```
un puntatore al testo della parola
un contatore del numero di occorrenze
un puntatore al nodo figlio di sinistra
un puntatore al nodo figlio di destra
```

Ogni nodo può avere zero, uno o due figli, ma non di più.

I nodi vengono ordinati in modo che, per ognuno di essi, il sottoalbero di sinistra contenga parole lessico-graficamente minori della parola contenuta nel nodo stesso, mentre il sottoalbero di destra contiene soltanto parole maggiori di quest'ultima.

Nella figura che segue è illustrato l'albero corrispondente alla frase "now is the time for all good men to come to the aid of their party", costruito inserendo ogni parola non appena è stata letta:

Per scoprire se una parola si trova già nell'albero, iniziamo dalla radice e confrontiamo la nuova parola con quella contenuta in quel nodo. Se esse sono uguali, la risposta al nostro quesito è affermativa. Se la nuova parola è minore di quella contenuta nel nodo in esame, la ricerca prosegue nel sottoalbero di sinistra, altri-menti in quello di destra. Se, nella direzione in cui dovrebbe proseguire la ricerca, non esistono ulteriori no-di, la nuova parola non è nell'albero, e la posizione vuota trovata è proprio quella in cui essa dev'essere in-serita. Questo procedimento è ricorsivo, poiché la ricerca su ogni nodo è legata a quella su uno dei suoi figli.

Di conseguenza, le funzioni più adatte per l'inserimento e la stampa di elementi dell'albero sono di tipo ricorsivo.

Tornando alla descrizione di un nodo, possiamo dire che esso è ben rappresentato da una struttura avente quattro componenti:

```
struct tnode {                  /* il nodo dell'albero */
    char *word;                 /* punta al testo */
    int count;                  /* numero di occorrenze */
    struct tnode  *left;        /* figlio di sinistra */
    struct tnode  *right;       /* figlio di destra */
};
```

La dichiarazione ricorsiva di un nodo può sembrare pericolosa, ma è corretta. Non è possibile dichiarare una struttura che contenga un'istanza di se stessa, ma

```
struct tnode  *left;
```

dichiara che `left` è un puntatore ad un oggetto `tnode`, non un `tnode` stesso.

A volte, può essere necessario utilizzare una variante delle strutture ricorsive: due strutture che si riferisco-no l'una all'altra. Il modo per gestire questi oggetti è il seguente:

```
struct t {
    ....
    struct s  *p;       /* p punta ad un oggetto di tipo s */
};
struct s {
    ....
    struct t  *q;       /* q punta ad un oggetto di tipo t */
};
```

Utilizzando routine di supporto già scritte, come `getword`, il codice per l'intero programma risulta sorprendentemente breve. La routine principale legge una parola con `getword` e la inserisce nell'albero tramite la funzione `addtree`.

```
#include  <stdio.h>
#include  <ctype.h>
#include  <string.h>

#define  MAXWORD  100
struct tnode  *addtree(struct tnode *, char *);
void  treeprint(struct tnode *);
int  getword(char *, int);

/* conta la frequenza delle parole */
main()
{
    struct tnode  *root;
    char word[MAXWORD];

    root=NULL;
    while (getword(word, MAXWORD)!=EOF)
        if (isalpha(word[0]))
            root=addtree(root, word);
    treeprint(root);
    return 0;
}
```

La funzione `addtree` è ricorsiva. La funzione `main` fornisce una parola ponendola al livello più alto (la radice) dell'albero. Ad ogni passo, la parola viene confrontata con quella già memorizzata nel nodo, e viene fat-ta scendere verso il ramo di sinistra o di destra tramite una chiamata ricorsiva ad `addtree`. Alla fine, o la parola risulta uguale ad una già presente nell'albero (ed in tal caso il contatore viene incrementato), oppure si incontra un puntatore nullo, che indica che all'albero dev'essere aggiunto un nodo creato appositamente per

la parola in esame. Se viene creato un nuovo nodo, `addtree` ritorna un puntatore ad esso, e tale pun-tatore viene memorizzato nel nodo padre.

```
struct tnode  *talloc(void);
char  *strdup(char *);

/* addtree: aggiunge un nodo con w, a livello di *p o sotto */
struct tnode  *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p==NULL)              /* ha ricevuto una nuova parola */
    {
        p=talloc();           /* crea un nuovo nodo */
        p->word=strdup(w);
        p->count=1;
        p->left=p->right=NULL;
    }
    else if ((cond=strcmp(w, p->word))==0)
        p->count++;           /* parola ripetuta */
    else if (cond<0)          /* minore: sottoalbero sinistro */
        p->left=addtree(p->left, w);
    else                      /* maggiore: sottoalbero destro */
        p->right=addtree(p->right, w);
    return p;
}
```

La memoria per il nuovo nodo viene allocata dalla routine `talloc`, che ritorna un puntatore allo spazio sufficiente per contenere il nodo stesso, e la nuova parola viene copiata da `strdup` in una posizione non nota (discuteremo queste funzioni in un secondo tempo). Il contatore viene inizializzato, ed i puntatori vengono posti a `NULL`. Questa parte del codice è eseguita soltanto sulle foglie dell'albero, quando si deve aggiunge-re un nuovo nodo. Abbiamo (imprudentemente) tralasciato il controllo degli errori sui valori restituiti da `tal-loc` e `strdup`.

`treeprint` stampa l'albero in modo ordinato; ad ogni nodo, la funzione stampa dapprima il sottoalbero di sinistra (tutte le parole minori di quella nel nodo in esame), quindi la parola stessa, ed infine il sottoalbero di destra (tutte le parole maggiori). Se non vi è chiaro il modo in cui lavora la ricorsione, simulate la funzione `treeprint` sull'albero mostrato in precedenza.

```
/* treeprint: stampa in-order dell'albero p */
void  treeprint(struct tnode *p)
{
    if (p!=NULL)
    {
        treeprint(p->left);
        printf("%4d %s\n", p->word);
        treeprint(p->right);
    }
}
```

Una nota pratica: se l'albero diventa "sbilanciato" perché le parole non arrivano in ordine casuale, il tempo di esecuzione del programma può crescere eccessivamente. Nel caso pessimo, se le parole sono già ordi-nate, questo programma non è altro che una costosa simulazione della ricerca lineare. Esistono delle generalizzazioni dell'albero binario che non sono soggette a questo peggioramento, ma noi non le descriveremo.

Prima di abbandonare definitivamente questo esempio, è utile fare una breve digressione sui problemi lega-ti agli allocatori di memoria. È ovviamente desiderabile che in un programma esista un unico allocatore di memoria, in grado di allocare diversi tipi di oggetti. Tuttavia, il fatto che un allocatore debba gestire, per esempio, l'allocazione di memoria sia per dei puntatori a `char` che per dei puntatori a `struct tnode`, solleva due problema. Innanzitutto, come può esso rispettare il requisito, presente su molte macchine, che oggetti di particolari tipi debbano soddisfare restrizioni di allineamento (per esempio, spesso gli interi devono essere collocati ad un indirizzo pari) ? In secondo luogo, quali dichiarazioni consentono di gestire il fatto che l'allocatore debba restituire puntatori ad oggetti di volta in volta differenti?

In genere, i requisiti di allineamento possono essere soddisfatti facilmente, a prezzo di un certo spreco di spazio, assicurandosi che l'allocatore ritorni sempre un puntatore che rispetta tutte le restrizioni sull'allineamento. La funzione `alloc` del Capitolo 5 non garantisce alcun allineamento particolare, quindi noi utilizzeremo la funzione della libreria standard `malloc`, che invece lo fa. Nel Capitolo 8 illustreremo una possibile implementazione di `malloc`.

Il problema della dichiarazione di tipo per una funzione come `malloc` affligge tutti i linguaggi che prendono in seria considerazione il controllo sui tipi. In C, la soluzione corretta consiste nel dichiarare che l'oggetto ritornato da `malloc` è un puntatore a `void`, e forzare poi esplicitamente questo puntatore a puntare al tipo desiderato, usando l'operatore cast. `malloc` e le routine ad essa legate sono dichiarate nell'header standard `<stdlib.h>`. Quindi, `talloc` può essere scritta nel modo seguente:

```
#include  <stdlib.h>

/* talloc: crea un tnode */
struct tnode  *talloc(void)
{
    return (struct tnode *)malloc(sizeof(struct tnode));
}
```

`strdup` non fa altro che copiare la stringa datale come argomento in un luogo sicuro, ottenuto con una chiamata a `malloc`:

```
char  *strdup(char *s)                    /* crea una copia di s */
{
    char *p;

    p=(char *)malloc(strlen(s)+1);        /* +1 per '\0' */
    if (p!=NULL)
        strcpy(p, s);
    return p;
}
```

`malloc` ritorna `NULL` se lo spazio richiesto non è disponibile; `strdup` passa all'esterno questo valore, lasciando al chiamante il compito di gestire l'errore.

La memoria ottenuta tramite la chiamata a `malloc` può essere liberata, invocando `free`, in vista di un impiego successivo; si vedano, a questo proposito, i Capitoli 7 e 8.

Esercizio 6.2 Scrivete un programma che legga un programma C e stampi in ordine alfabetico ogni gruppo di nomi di variabili identici per i primi sei caratteri e diversi in qualcuno dei successivi. Non contate le parole all'interno di stringhe e commenti. Fate in modo che 6 sia un parametro, stabilito dalla linea di comando.

Esercizio 6.3 Scrivete un programma di riferimenti incrociati ("cross-referencer") che stampi una lista di tut-te le parole presenti in un documento e, per ogni parola, una lista dei numeri di linea nei quali la parola ri-corre. Eliminate congiunzioni, articoli e così via.

Esercizio 6.4 Scrivete un programma che stampi le diverse parole del suo input, ordinandole in modo decrescente oppure in base alla loro frequenza di occorrenza. Precedete la stampa di ogni parola con il numero di occorrenze.

6.6 Analisi delle Tabelle

In questa sezione descriveremo a grandi linee un pacchetto di funzioni per l'analisi delle tabelle, con lo scopo di illustrare altre caratteristiche delle strutture. Questo codice è quello che, tipicamente, viene utilizzato nelle funzioni di gestione della tabella dei simboli (symbol table) presenti in un compilatore od in un macroprocessore. Per esempio, consideriamo l'istruzione `#define`. Quando una linea del tipo

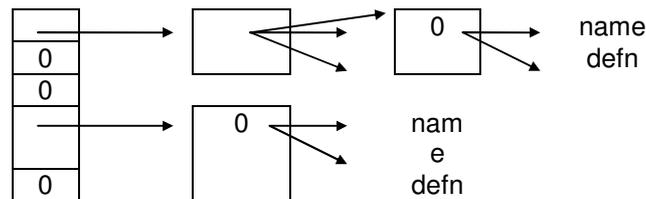
```
#define  IN  1
```

viene incontrata, nella tabella vengono memorizzati il nome `IN` ed il testo da sostituire, `1`. In seguito, quan-do il nome `IN` compare in un'istruzione come

```
state=IN;
```

esso dev'essere rimpiazzato con `1`.

Esistono due funzioni che manipolano i nomi ed i testi da sostituire. `install(s, t)` memorizza il nome `s` ed il testo `t` in una tabella; `s` e `t` sono delle stringhe di caratteri. `lookup(s)` cerca `s` nella tabella, e restituisce un puntatore alla posizione nella quale `s` si trova, oppure `NULL` se `s` non compare.

L'algoritmo è una ricerca hash: il nome in input viene convertito in un piccolo intero non negativo, che viene poi utilizzato come indice in un vettore di puntatori. Un elemento del vettore punta all'inizio di una lista concatenata di blocchi, che descrivono nomi aventi quel particolare valore di hash. Esso è NULL se nessuna parola ha quel valore di hash.

Un blocco della lista è una struttura che contiene puntatori al nome, al testo da sostituire ed al blocco successivo nella lista. Se quest'ultimo puntatore è nullo, la lista è finita.

```
struct nlist                 /* elemento della tabella */
{
    struct nlist  *next;     /* prossimo elemento della catena */
    char *name;              /* nome definito */
    char *defn;              /* testo da sostituire */
}
```

Il vettore di puntatori è, semplicemente,

```
#define  HASHSIZE  101

static struct nlist  *hashtab[HASHSIZE];       /* tabella dei puntatori */
```

La funzione di hashing, che viene usata sia da `lookup` che da `install`, somma il valore di ogni chiamata della stringa ad una combinazione rimescolata dei caratteri precedenti, e ritorna il resto della divisione fra il valore finale così ottenuto e la dimensione del vettore. Questa non è la migliore funzione di hashing possibile, ma è breve ed efficace.

```
/* hash: calcola il valore di hashing della stringa s */
unsigned  hash(char *s)
{
    unsigned hashval;

    for (hashval=0; *s!='\0'; s++)
        hashval=*s+31*hashval;
    return hashval % HASHSIZE;
}
```

L'aritmetica degli oggetti privi di segno assicura che il valore di hash sia non negativo.

Il processo di hashing produce un indice di partenza del vettore `hashtab`; se la stringa si trova da qualche parte nella tabella, essa sarà nella lista di blocchi che inizia da quell'indice. La ricerca viene effettuata da `lookup`. Se `lookup` trova che la parola è già presente, ritorna un puntatore ad essa, altrimenti ritorna NULL.

```
/* lookup: cerca s in hashtab */
struct nlist  *lookup(char *s)
{
    struct nlist  *np;
```

```
    for (np=hashtab[hash(s)]; np!=NULL; np=np->next)
        if (strcmp(s, np->name)==0)
            return p;                       /* trovata */
    return NULL;                            /* non trovata */
}
```

Il ciclo `for` presente in `lookup` rappresenta l'idioma classico utilizzato per attraverso una lista concatenata:

```
for (ptr=head; ptr!=NULL; ptr++)
    ....
```

`install` usa `lookup` per determinare se il nome in esame è già presente; se lo è, la nuova versione annulla la precedente. Altrimenti, viene creato un nuovo elemento. `install` ritorna `NULL` se, per un motivo qualsiasi, lo spazio per un nuovo elemento non è disponibile.

```
struct nlist  *lookup(char *s)
char  *strdup(char *);

/* install: inserisce (name, defn) in hashtab */
struct nlist  *install(char *name, char *defn)
{
    struct nlist  *np;
    unsigned hashval;

    if ((np=lookup(name))==NULL)          /* non trovata */
    {
        np=(struct nlist *)malloc(sizeof(*np));
        if (np==NULL  ||  (np->name=strdup(name))==NULL)
            return NULL;
        hashval=hash(name);
        np->next=hashtab[hashval];
        hashtab[hashval]=np;
    }
    else                                  /* name è già in hashtab */
        free((void *)np->defn);           /* annulla il precedente defn */
    if ((np->defn=strdup(defn))==NULL)
        return NULL;
    return np;
}
```

Esercizio 6.5 Scrivete una funzione `undef`, che rimuova un nome e la sua corrispondente definizione dalla tabella creata e gestita da `lookup` e `install`.

Esercizio 6.6 Implementate una semplice versione del processore di `#define` (non considerate le macro con argomenti), che sia utile per i programmi C; basatevi sulle routine di questa sezione. Potreste trovare molto utili anche le funzioni `getch` e `ungetch`.

6.7 TYPEDEF

Per creare nuovi nomi di tipi di dati, il C fornisce uno strumento chiamato `typedef`. Per esempio, la dichiarazione

```
typedef int Lenght;
```

definisce `Lenght` come sinonimo di `int`. Il tipo `Lenght` può essere utilizzato nelle dichiarazioni, nei cast, ecc., esattamente come il tipo `int`:

```
Lenght len, maxlen;
Lenght *lenghts[];
```

Analogamente, la dichiarazione

```
typedef char *String;
```

definisce `String` come sinonimo di `char *`, cioè come un puntatore a carattere, che può venir impiegato nelle dichiarazioni e nei cast:

```
String p, lineptr[MAXLINES], alloc(int);
int  strcmp(String, String);
p=(String)malloc(100);
```

Notate che il nome dichiarato da una `typedef` appare nella posizione del nome di variabile, e non subito dopo la parola chiave `typedef`. Sintatticamente, `typedef` è analogo ad una classe di memoria come `extern`, `static`, ecc. Per evidenziare i tipi definiti tramite `typedef`, abbiamo utilizzato delle iniziali maiuscole.

Come esempio più complesso, usiamo `typdef` all'interno del programma che abbiamo scritto per gestire i nodi di un albero:

```
typedef struct tnode  *Treeptr;

typedef struct tnode {        /* il nodo dell'albero */
    char *word                /* punta al testo */
    int count                 /* numero di occorrenze */
    Treeptr left              /* figlio di sinistra */
    Treeptr right             /* figlio di destra */
} Treenode;
```

Questa dichiarazione crea due nuove parole chiave, che definiscono due tipi: `Treenode` (una struttura) e `Treeptr` (un puntatore ad una struttura). Ora, la routine `talloc` può diventare

```
Treeptr talloc(void)
{
    return (Treeptr)malloc(sizeof(Treenode));
}
```

È bene sottolineare che una dichiarazione `typedef` non crea esattamente un nuovo tipo; semplicemente, essa aggiunge un nuovo nome ad un tipo già esistente. Neppure dal punto di vista semantico questa dichiarazione gode di particolari proprietà: le variabili dichiarate in questo modo hanno le stesse caratteristiche di quelle dichiarate in modo esplicito. In effetti, `typedef` è analoga a `#define`, a parte il fatto che, essendo interpretata dal compilatore, la prima consente di gestire sostituzioni di testo che vanno oltre le capacità del preprocessore. Per esempio,

```
typedef int (*PFI)(char *, char *);
```

crea il tipo `PFI`, la cui sigla deriva da "puntatore a funzione (con due argomenti `char *`) che ritorna un `int`", che può essere utilizzato in contesti simili al seguente:

```
PFI strcmp, numcmp;
```

inserito nel programma di ordinamento del Capitolo 5.

A parte i motivi puramente estetici, due sono le ragioni che rendono auspicabile l'impiego di `typedef`. La prima è quella di parametrizzare il più possibile un programma, al fine di aumentarne la portabilità. Se, per i tipi dipendenti dall'architettura della macchina, vengono utilizzate delle `typedef`, spostando il programma sarà sufficiente modificare le `typedef` stesse. Una situazione molto comune è quella in cui `typedef` viene usata per dare nomi diversi a diverse quantità intere, creando così un insieme particolare di `short`, `int` e `long` per ogni macchina. Esempi di questo genere sono dati dai tipi `size_t` e `ptrdiff_t`, definiti nella libreria standard.

Il secondo scopo delle `typedef` è quello di fornire una migliore documentazione di un programma: il significato di un tipo chiamato `Treeptr` risulta più comprensibile di quello di un puntatore ad una struttura complessa.

6.8 UNION

Una *union* è una variabile che può contenere (in istanti diversi) oggetti di tipo e dimensioni differenti, dei quali il compilatore gestisce l'ampiezza in base ai requisiti di allineamento. Le union consentono di manipo-lare diversi tipi di dati all'interno di un'unica area di memoria, senza che questo implichi l'inserimento di in-formazioni dipendenti dalla macchina all'interno dei programmi. Esse sono analoghe ai record varianti del Pascal.

Come esempio di qualcosa che potrebbe essere trovato in un compilatore in relazione alla gestione della tabella dei simboli, assumiamo che una costante possa essere un `int`, un `float` od un puntatore a carat-tere. Il valore di una particolare costante dev'essere registrato in una variabile del tipo corretto, anche se, per il compilatore, è più conveniente poter inserire tale valore, indipendentemente dal suo tipo, sempre nel-la stessa area di memoria. Questo è lo scopo di una union: una singola variabile può contenere uno qual-siasi dei tipi facenti parte di un certo insieme. La sintassi si basa sulle strutture:

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

La variabile `u` è sufficientemente grande da contenere il più ampio dei tre tipi; la dimensione specifica è di-pendente dall'implementazione. Ognuno di questi tre tipi può essere assegnato alla variabile `u` ed utilizzato nelle espressioni, purché l'utilizzo sia consistente: il tipo impiegato dev'essere l'ultimo memorizzato. È re-sponsabilità del programmatore tenere traccia di quale sia il tipo attualmente registrato in una `union`; se ciò che è stato memorizzato è di tipo diverso da ciò che viene estratto, il risultato dell'operazione dipende dalla implementazione.

Sintatticamente, ai membri di una `union` si accede nel modo seguente:

```
nome-union.membro
```

oppure

```
puntatore-union->membro
```

analogamente a quanto si fa con le strutture. Se la variabile `utype` viene usata per tenere traccia del tipo correntemente memorizzato in `u`, allora codice simile a quello che segue può risultare frequente

```
if (utype==INT)
    printf("%d\n", u.ival);
else if (utype==FLOAT)
    printf("%f\n", u.fval);
else if (utype==STRING)
    printf("%s\n", u.sval);
else
    printf("Tipo %d scorretto in utype\n", utype);
```

Le union possono trovarsi all'interno di strutture e vettori, e viceversa. La notazione per accedere al mem-bro di una union in una struttura (o viceversa) è identica a quella usata per le strutture nidificate. Per esem-pio, nel vettore di strutture definito da

```
struct {
    char *name;
    int flags;
    int utype;
    union {
        int ival;
        float fval;
        char *sval;
    } u;
} symtab[NSYM];
```

Il membro `ival` può essere riferito come

```
symtab[i].u.ival
```

ed il primo carattere della stringa `sval` può essere ricavato con una delle due espressioni

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

In ultima analisi, una union è una struttura nella quale tutti i membri hanno spiazzamento nullo dalla base, la struttura è sufficientemente grande da potere contenere il membro "più ampio", e l'allineamento è corretto per tutti i tipi presenti nella union. Le operazioni consentite sulle union sono quelle consentite sulle struttu-re: l'assegnamento o la copia in blocco, il prelievo dell'indirizzo e l'accesso ai membri.

Una union può venire inizializzata fornendo soltanto il valore del tipo del suo primo membro; quindi, la union `u` sopra descritta può venire inizializzata con un valore intero.

L'allocatore di memoria illustrato nel Capitolo 8 mostra come una union possa essere impiegata per forzare l'allineamento di una variabile ad indirizzi multipli di un valore dipendente da un particolare tipo di memoria.

6.9 Campi di Bit

Quando lo spazio in memoria è scarso, può risultare necessario inserire diversi oggetti in un'unica word della macchina; una soluzione molto usata consiste nell'impiegare, in applicazioni come la gestione delle tabelle dei simboli nei compilatori, un insieme di flag di un singolo bit. Anche i formati dei dati imposti dallo esterno, come le interfacce hardware verso i device, richiedono spesso la capacità di gestire segmenti di una word.

Immaginiamo la parte di un compilatore relativa alla gestione della tabella dei simboli. Sicuramente, in un programma ogni identificatore ha delle informazioni associate quali, per esempio, il fatto che sia o meno una parola chiave, il fatto che sia o meno esterno e/o static, e così di seguito. Il modo più compatto di codifi-care queste informazioni è un insieme di flag di un bit, contenuti in un singolo `char` o `int`.

Il modo più frequente per realizzare una simile soluzione consiste nel definire un insieme di "maschere" che corrispondono alle posizioni dei bit significativi, come in

```
#define  KEYWORD   01
#define  EXTERNAL  02
#define  STATIC    04
```

oppure

```
enum {KEYWORD=01, EXTERNAL=02, STATIC=04};
```

I numeri devono essere potenze di due. In questo modo, l'accesso ai bit si riduce ad un problema di "sintonizzazione di bit", risolvibile con gli operatori di shift, di masking e di completamento illustrati nel Capitolo 2.

Esistono alcuni idiomi molto comuni:

```
flags |= EXTERNAL | STATIC;
```

alza, in `flags`, i bit `EXTERNAL` e `STATIC`, mentre

```
flags &= ~(EXTERNAL | STATIC);
```

li abbassa, e

```
if ((flags & (EXTERNAL | STATIC))==0)  ....
```

è vero se entrambi i bit sono nulli.

Sebbene questi idiomi siano abbastanza chiari, il C offre la possibilità di definire ed accedere ai campi allo interno di una word in modo diretto, invece che attraverso gli operatori logici bit a bit. Un *bit-field*, o semplicemente *field*, è un insieme di bit adiacenti all'interno di una singola unità di memoria, definita dall'implementazione, che chiameremo "word". La sintassi della definizione di un field e del riferimento ad esso si basa sulle strutture. Per esempio, le tre `#define` precedenti potrebbero essere sostituite dai tre field:

```
struct {
    unsigned int is_keyword:  1;
    unsigned int is_extern:   1;
    unsigned int is_static:   1;
} flags;
```

Questa dichiarazione definisce una variabile `flags`, che contiene tre field di un bit. Il numero che segue il due punti rappresenta l'ampiezza del field. I field vengono dichiarati come `unsigned int`, per assicurare che essi siano sempre quantità prive di segno.

Ai singoli field si accede in modo analogo a quanto avviene per i membri delle strutture: `flags.is_keyword`, `flags.is_extern` ecc. I field si comportano come dei piccoli interi, e possono intervenire in espressioni aritmetiche, esattamente come gli altri interi. Quindi, gli esempi precedenti possono essere riscritti come:

```
flags.is_extern=flags.is_static=1;
```

per alzare i due bit;

```
flags.is_extern=flags.is_static=0;
```

per abbassarli; e

```
if (flags.is_extern==0  && flags.is_static==0)
    ....
```

per controllarli.

Quasi tutto ciò che riguarda i field è dipendente dalla macchina. Il fatto che un field possa trovarsi al confine fra due word successive dipende dalla macchina. I field possono non avere nome; quelli privi di nome (un due punti seguito da un'ampiezza) vengono usati per colmare i "buchi". L'ampiezza speciale 0 può essere utilizzata per forzare un allineamento alla word successiva.

I field vengono assegnati da sinistra a destra su alcune macchine, e da destra a sinistra su altre. Questo significa che, anche se essi sono molto utili per mantenere delle strutture dati definite internamente, il problema di quale sia l'estremo che viene raggiunto per primo dev'essere considerato con molta attenzione, quando si prelevano dati definiti esternamente; i programmi che dipendono da queste caratteristiche non sono portabili. I field possono essere dichiarati soltanto come `int`; per motivi di portabilità, è necessario specificare esplicitamente se sono `signed` o `unsigned`. Essi non sono dei vettori, e non possiedono indirizzo, quindi ad essi non può essere applicato l'operatore `&`.

CAPITOLO 7

INPUT ED OUTPUT

Le funzionalità di input ed output non fanno parte del linguaggio C, e per questo motivo, fino a questo momento, non le abbiamo sottolineate in modo particolare. Tuttavia, i programmi interagiscono con il loro ambiente in modo notevolmente più complesso di quanto non abbiamo mostrato sino ad ora. In questo capitolo descriveremo la libreria standard, un insieme di funzioni che consentono l'input / output, la gestione delle stringhe e della memoria, le funzioni matematiche e diversi altri servizi utili nei programmi C. In particolare, concentreremo la nostra attenzione sull'input / output.

Lo standard ANSI definisce in modo preciso queste funzioni di libreria, cosicché esse siano compatibili su tutti i sistemi che supportano il C. I programmi che limitano le loro interazioni con il sistema all'impiego delle routine dalla libreria standard, possono essere trasportati da una macchina all'altra senza alcuna modifica.

Le proprietà delle funzioni di libreria sono specificate in una dozzina di header, come ad esempio <stdio.h>, <string.h> e <ctype.h>, che abbiamo già incontrato. In questa sede non presenteremo l'intera libreria, poiché siamo più interessati alla stesura di programmi che la utilizzino. La libreria viene descritta in dettaglio nell'Appendice B.

7.1 Input ed Output Standard

Come abbiamo detto nel Capitolo 1, la libreria implementa un modello molto semplice di input ed output riguardante i testi. Un flusso di testo consiste in una sequenza di linee; ogni linea termina con un carattere di new line. Se il sistema non opera in questo modo, la libreria fa tutto ciò che è necessario affinché appaia il contrario. Per esempio, la libreria può convertire i caratteri di return e di linefeed in caratteri di new line sia quando li riceve in input che quando li pone in output.

Il più semplice meccanismo di input consiste nel leggere un carattere per volta dallo *standard input*, normalmente la tastiera, usando getchar:

```
int  getchar(void)
```

ad ogni chiamata, getchar restituisce il successivo carattere in input, oppure il valore EOF se incontra la fine del file. La costante simbolica EOF è definita in <stdio.h>. Il suo valore è, in genere, -1, ma tutti i controlli dovrebbero utilizzare il termine EOF, in modo da risultare indipendenti dal valore specifico.

In molti ambienti la tastiera può essere sostituita con un file qualsiasi, utilizzando la convenzione < di redirezione dell'input: se il programma prog usa getchar, allora la linea di comando

```
prog <infile
```

costringe prog a leggere i caratteri dal file infile. La sostituzione dell'input viene effettuata in modo da non interessare direttamente prog; in particolare, la stringa "<infile" non fa parte degli argomenti della linea di comando, registrati in argv. La trasparenza viene mantenuta anche nel caso in cui l'input sia fornito da un altro programma, attraverso il meccanismo di pipe: su alcuni sistemi, la linea di comando

```
otherprog | prog
```

comporta l'esecuzione dei programmi otherprog e prog, e sostituisce lo standard input di prog con lo standard output di otherprog.

La funzione

```
int  putchar(int)
```

viene utilizzata per l'output: putchar(c) invia il carattere c allo *standard output*, che normalmente è il video; *putchar ritorna il carattere scritto, oppure EOF se si verifica un errore. Ancora, l'output può venire facilmente rediretto su un file usando >nomefile*: se prog usa putchar,

```
prog >outfile
```

scriverà su outfile invece che sullo standard output. Se il meccanismo di pipe è supportato,

```
prog | anotherprog
```

sostituisce lo standard output di prog allo standard input di anotherprog.

Anche l'output prodotto da printf viene inviato allo standard output. Le chiamate a putchar e printf possono essere intercalate: l'output appare nell'ordine in cui le chiamate sono state effettuate.

Ogni file sorgente che utilizza una qualsiasi funzione di input / output della libreria standard deve contenere la linea

```
#include  <stdio.h>
```

prima di quella in cui la funzione viene usata per la prima volta. Quando il nome dell'header è racchiuso fra < e >, la ricerca dell'header stesso avviene in un insieme standard di directory (per esempio, sui sistemi UNIX, la directory /usr/include).

Molti programmi si limitano a leggere un unico flusso di input e a scriverne un altro; per tali programmi, l'input e l'output realizzabili con le funzioni getchar, putchar e printf possono rivelarsi del tutto adeguati, e sono certamente sufficienti per i primi approcci di un programmatore inesperto. Ciò è particolarmente vero se si usa la redirezione per connettere l'output di un programma con l'input del successivo. Per esempio, consideriamo il programma lower, che converte il suo input in lettere minuscole:

```
#include  <stdio.h>
#include  <ctype.h>

main()      /* lower: converte l'input in lettere minuscole */
{
    int c;

    while ((c=getchar())!=EOF)
        putchar(tolower(c));
    return 0;
}
```

La funzione tolower è definita in <ctype.h>; essa converte una lettera maiuscola nel suo corrispondente minuscolo, e restituisce invariati tutti gli altri caratteri. Come abbiamo già detto, le "funzioni" come getchar e putchar in <stdio.h> e come tolower in <ctype.h> sono spesso delle macro, il che consente di evitare l'overhead che si avrebbe se al trattamento di ogni carattere corrispondesse una chiamata di funzione. Nel Capitolo 8 illustreremo come è possibile realizzare tutto ciò. Indipendentemente da come vengono implementate le funzioni di <ctype.h> su una particolare macchina, i programmi che le utilizzano possono ignorare quale sia il set di caratteri adottato.

Esercizio 7.1 Scrivete un programma che converta la lettere maiuscole in minuscole e viceversa, in base al nome con il quale viene invocato, registrato in argv[0].

7.2 Output Formattato - PRINTF

La funzione di output printf traduce dei valori interni in caratteri. Nei capitoli precedenti abbiamo usato printf in modo informale. La descrizione che ne diamo in questo paragrafo, pur non essendo completa, copre i casi più comuni; per maggiori dettagli, si veda l'Appendice B.

```
int  printf(char *format, arg1, arg2, ....)
```

printf converte, formatta e stampa sullo standard output i suoi argomenti sotto il controllo di format. Essa ritorna il numero di caratteri stampati.

La stringa di formato contiene due tipi di oggetti: caratteri ordinari, che vengono semplicemente copiati sullo output, e specifiche di conversione, ognuna delle quali provoca la conversione e la stampa del successivo

argomento di printf. Ogni specifica di conversione inizia con un % e termina con un carattere di conversione. Tra % ed il carattere di conversione possiamo trovare, nell'ordine:

a) Un segno meno, che specifica l'allineamento a sinistra dell'argomento convertito.
b) Un numero che specifica l'ampiezza minima del campo. L'argomento convertito viene stampato in un campo di ampiezza almeno pari a quella data. Se necessario, vengono lasciati degli spazi bianchi a sinistra (o a destra, se è stato richiesto l'allineamento a sinistra) che consentono di raggiungere l'ampiezza desiderata.
c) Un punto, che separa l'ampiezza del campo dalla precisione.
d) Un numero, la precisione, che specifica il massimo numero di caratteri che devono essere stampati da una stringa, oppure il numero di cifre dopo il punto decimale di un numero floating-point, oppure ancora il numero minimo di cifre di un intero.
e) Una h se l'intero dev'essere stampato come short, oppure l (lettera elle) se dev'essere stampato come long.

I caratteri di conversione sono illustrati in Tabella 7.1. Se il carattere che segue % non è una specifica di conversione, il comportamento di printf è indefinito.

Tabella 7.1 *Conversioni base di PRINTF.*

CARATTERE	TIPO DELL'ARGOMENTO; STAMPATO COME
d, i	int; numero decimale.
o	int; numero ottale privo di segno (senza zero iniziale).
x, X	int; numero esadecimale privo di segno (senza 0x o 0X iniziale), stampato usando abcdef o ABCDEF per 10, ..., 15.
U	int; numero decimale privo di segno.
c	int; carattere singolo.
S	char *; stampa caratteri dalla stringa fino al raggiungimento di '\0' o della precisione.
f	double; [-]*m.dddddd*, dove il numero delle *d* è dato dalla precisione (il default è 6).
e, E	double; [-]*m.dddddd*e±*xx* oppure [-]*m.dddddd*E±*xx*, dove il numero delle *d* è dato dalla precisione (il default è 6).
g, G	double; usa %e o %E se l'esponente è minore di -4 o maggiore o uguale alla precisione; altrimenti usa %f. Gli zeri superflui non vengono stampati.
p	void *; puntatore (rappresentazione dipendente dall'implementazione).
%	non converte alcun argomento; stampa un %.

Un'ampiezza o precisione può essere specificata come *, nel qual caso il valore viene calcolato converten-do l'argomento successivo, che dev'essere un int. Per esempio, per stampare al più max caratteri di una stringa s, possiamo scrivere

```
printf("%.*s", max, s);
```

La maggior parte dei formati di conversione è stata illustrata nei capitoli precedenti. Un'eccezione è data dalla precisione in relazione alle stringhe. La tabella seguente illustra l'effetto di un certo numero di specifi-che diverse usate per stampare "salve, mondo" (12 caratteri). Intorno ad ogni campo abbiamo posto dei due punti, in modo che possiate vederne l'ampiezza.

```
:%s:            :salve, mondo:
:%10s:          :salve, mondo:
:%.10s:         :salve, mon:
:%-10s:         :salve, mondo:
:%.15s:         :salve, mondo:
:%-15s:         :salve, mondo   :
:%15.10s:       :     salve, mon:
:%-15.10s:      :salve, mon     :
```

Un avvertimento: printf usa il suo primo argomento per decidere quanti e di che tipo sono gli argomenti successivi. Se gli argomenti non sono sufficienti o se sono di tipo scorretto, printf produce stampe diver-se da quelle che voi vi attendereste. Dovete anche prestare molta attenzione alla differenza che esiste fra le seguenti due chiamate:

```
printf(s);              /* Fallisce se s contiene qualche % */
printf("%s", s);        /* CORRETTO */
```

La funzione `sprintf` effettua conversioni uguali a `printf`, ma memorizza il suo output in una stringa:

```
int  sprintf(char *string, char *format, arg1, arg2, ....)
```

`sprintf` formatta, in base a `format`, gli argomenti *arg1*, *arg2* ecc., ma pone il risultato in `string`, invece che sullo standard output; `string` dev'essere sufficientemente ampia da potere contenere il risultato.

Esercizio 7.2 Scrivete un programma che stampa in modo sensato un input arbitrario. Come requisito minimo, esso deve scrivere caratteri non stampabili in ottale o in esadecimale, in base all'uso più comune, e spezzare le linee troppo lunghe.

7.3 Liste di Argomenti di Lunghezza Variabile

Questa sezione contiene un'implementazione di una versione minimale di `printf`, che ha lo scopo di illustrare come scrivere in modo portabile una funzione che ha una lista di argomenti di lunghezza variabile. Poiché siamo interessati soprattutto al trattamento degli argomenti, facciamo in modo che `minprintf` gestisca la stringa di formato ed i suoi argomenti, ma richiami `printf` per eseguire le conversioni di formato.

La dichiarazione corretta di `printf` è:

```
int  printf(char *fmt, ...)
```

dove la dichiarazione `...` significa che il numero ed i tipi degli argomenti che seguono sono variabili. La dichiarazione `...` può comparire soltanto al termine di una lista di argomenti. La nostra funzione `minprintf` è dichiarata come

```
void  minprintf(char *fmt, ...)
```

perché, al contrario di `printf`, essa non restituisce il numero di caratteri trattati.

Degno di particolare attenzione è il modo in cui `minprintf` scandisce la lista dei suoi argomenti, dei quali non conosce neppure il nome. L'header standard `<stdarg.h>` contiene un insieme di macro che definiscono come scandire una lista di argomenti. L'implementazione di questo header varia da macchina a macchina, ma l'interfaccia che esso presenta è unica.

Per dichiarare una variabile che si riferisca, in momenti diversi, ai diversi argomenti, viene usato il tipo `va_list`; in `minprintf`, questa variabile viene chiamata `ap`, da "argument pointer (puntatore all'argomento)". La macro `va_start` inizializza `ap` in modo che punti al primo argomento privo di nome, e dev'essere chiamata prima dell'utilizzo di `ap`. Deve sempre esistere almeno un argomento con un nome, l'ultimo dei quali viene usato da `va_start` per l'inizializzazione.

Ogni chiamata a `va_arg` restituisce un argomento ed incrementa `ap` in modo che punti al successivo; `va_arg` usa un nome di tipo per determinare quale tipo ritornare e di quanto incrementare il puntatore. Infine, `va_end` ripulisce tutto ciò che è necessario. Essa dev'essere chiamata prima di uscire dalla funzione.

Queste proprietà costituiscono le basi sulle quali si fonda la nostra `printf` semplificata:

```
#include  <stdarg.h>

/* minprintf: printf amplificata con una lista di argomenti variabile */
void  minprintf(char *fmt, ...)
{
    va_list ap;             /* punta, a turno, tutti gli argomenti
                               privi di nome */
    char *p, *sval;
    int ival;
    double dval;
```

```
    va_start(ap, fmt);          /* fa puntare ap al primo argomento
                                senza nome */
    for (p=fmt; *p; p++)
    {
        if (*p!='%')
        {
            putchar(*p);
            continue;
        }
        switch (*++p) {
        case 'd':
            ival=va_arg(ap, int);
            printf("%d", ival);
            break;
        case 'f':
            dval=va_arg(ap, double);
            printf("%f", dval);
            break;
        case 's':
            for (sval=va_arg(ap, char *); *sval; sval++)
                putchar(*sval);
            break;
        default:
            putchar(*p);
            break;
        }
    }
    va_end(ap);                 /* ripulisce quando ha finito */
}
```

Esercizio 7.3 Riscrivete `minprintf` in modo che gestisca qualcuna delle altre funzionalità fornite da `printf`.

7.4 Input Formattato - SCANF

La funzione `scanf` è l'analogo, ma per l'input, di `printf`; essa fornisce infatti, nella direzione opposta, molte delle possibilità di conversione offerte da `printf`.

```
int  scanf(char *format, ...)
```

`scanf` legge caratteri dallo standard input, li interpreta in base al contenuto della stringa `format` e memorizza il risultato di queste operazioni negli argomenti successivi. L'argomento `format` verrà descritto nel seguito di questa sezione; gli altri argomenti, *ognuno dei quali dev'essere un puntatore*, indicando dove registrare l'input convertito. Come per `printf`, questa sezione vuole essere un riassunto delle funzionalità più utili, e non un loro elenco esaustivo.

`scanf` termina quando esaurisce la sua stringa di formato, oppure quando riscontra un'inconsistenza fra lo input e le specifiche di controllo. Il suo valore di ritorno è il numero di elementi in input letti e registrati correttamente, ed in quanto tale può essere utilizzato per determinare il numero di oggetti trovati. Al termine del file, `scanf` ritorna il valore `EOF`; notate che questo valore è diverso dallo 0, che viene invece restituito quando il carattere in input è in contrasto con la prima specifica di controllo presente in `format`. Ogni chia-mata a `scanf` riprende dal carattere immediatamente successivo all'ultimo convertito.

Esiste anche una funzione, `sscanf`, che legge da una stringa invece che dallo standard input:

```
int  sscanf(char *string, char *format, arg1, arg2, ....)
```

`sscanf` scandisce la stringa `string` in base alle specifiche date in `format`, e memorizza in arg_1, arg_2 ecc. (che devono essere puntatori) i valori risultanti.

In genere, la stringa di formato contiene alcune specifiche di conversione, utilizzate per il controllo della conversione dell'input. Tali specifiche possono contenere:

a) Spazi o caratteri di tabulazione, che vengono ignorati.
b) Caratteri normali (non %), che ci si aspetta corrispondano al successivo carattere non bianco del flusso di input.
c) Specifiche di conversione, costituite dal carattere %, da un * opzionale di soppressione dell'assegnamento, da un numero opzionale che specifica la massima ampiezza del campo, da un h, l o L opzionali indicanti la dimensione dell'elemento, e da un carattere di conversione.

Una specifica di conversione si riferisce alla conversione del successivo campo in input. Normalmente il risultato viene registrato nella variabile alla quale punta l'argomento corrispondente. Tuttavia, se è presente il carattere * ad indicare la soppressione dell'assegnamento, il campo in input viene ignorato; non viene effettuato alcun assegnamento. Un campo in input è definito come una stringa priva di caratteri di spaziatura; esso si estende fino al primo carattere di spaziatura oppure fino all'ampiezza massima, se specificata. Questo implica che scanf cerchi il suo input all'interno di più linee, perché i new line sono considerati semplici caratteri di spaziatura (i caratteri di spaziatura sono gli spazi, i new line, i return, i tab orizzontali e verticali ed i salti pagina).

Il carattere di conversione indica l'interpretazione del campo in input. L'argomento corrispondente dev'essere un puntatore, com'è richiesto dalla semantica della chiamata per valore del C. I caratteri di conversione sono illustrati in Tabella 7.2.

I caratteri di conversione d, i, o, u e x possono essere preceduti da h, per indicare nella lista di argomenti, invece di un puntatore ad un int, compare un puntatore ad uno short; oppure, essi possono essere preceduti da l (lettera elle) per indicare che quello che compare è un puntatore ad un long. Analogamente, i caratteri di conversione e, f e g possono essere preceduti da l per indicare che nella lista degli argomenti compare un puntatore ad un double invece che ad un float.

Come primo esempio possiamo riscrivere, usando scanf, la rudimentale calcolatrice del Capitolo 4.

```
#include  <stdio.h>

main()            /* rudimentale calcolatrice */
{
    double sum, v;

    sum=0;
    while (scanf("%lf", &v)==1)
        printf("\t%.2f\n", sum+=v);
    return 0;
}
```

Supponiamo di volere leggere linee di input contenenti date espresse nella forma

```
25 Dec 1988
```

Tabella 7.2 *Conversioni base di SCANF*

CARATTERE	DATI IN INPUT; TIPO DELL'ARGOMENTO
d	intero decimale; int *.
i	intero; int *. L'intero può essere in ottale (preceduto da uno 0) oppure in esadecimale (preceduto da 0x o 0X).
o	intero ottale (preceduto o meno dallo 0); int *.
x	intero esadecimale (preceduto o meno da 0x o 0X); int *.
c	caratteri; char *. I prossimi caratteri in input (1 per default) vengono inseriti nella posizione indicata; vengono considerati anche i caratteri di spaziatura; per leggere il prossimo carattere non bianco, usate %1s.
s	stringa di caratteri (non racchiusa fra apici); char *, che punta ad un vettore di caratteri sufficientemente grande da contenere la stringa ed uno '\0' di terminazione, che verrà aggiunto automaticamente.
e, f, g	numero floating-point con segno, punto decimale ed esponente opzionali; float *.
%	carattere %; non viene effettuato alcun assegnamento.

L’istruzione `scanf` è la seguente

```
int day, year;
char monthname[20];

scanf(“%d %s %d”, &day, monthname, &year);
```

`monthname`, essendo un nome di vettore e quindi un puntatore, non ha bisogno di essere preceduto da `&`.

Nella stringa di formato di `scanf` possono comparire caratteri normali; essi, però, devono corrispondere ai caratteri dell’input. Di conseguenza, usando `scanf` potremmo leggere date espresse nella forma `mm/dd /yy`:

```
int day, month, year;

scanf(“%d/%d/%d”, &month, &day, &year);
```

`scanf` ignora i caratteri di spaziatura, sia nella sua stringa di formato che nell’input. Per leggere un input il cui formato non è fissato, è spesso meglio leggere una linea per volta, la quale verrà trattata separatamente tramite `sscanf`. Per esempio, supponiamo di volere leggere linee contenenti date che possono essere espresse in una qualsiasi delle forme esposte in precedenza. Potremmo allora scrivere:

```
while (getline(line, sizeof(line))>0)
{
    if (sscanf(line, “%d %s %d”, &day, monthname, &year)==3)
                  /* forma 25 Dec 1988 */
        printf(“Input corretto: %s\n”, line);
    else if (sscanf(line, “%d/%d/%d”, &day, &month, &year)==3)
                  /* forma 25/12/1988 */
        printf(“Input corretto: %s\n”, line);
    else
        printf(“Input scorretto: %s\n”, line);
}
```

Le chiamate a `scanf` possono essere intercalate a chiamate di altre funzioni di input. Ogni chiamata a qualsiasi funzione di input inizierà leggendo il primo carattere non letto da `scanf`.

Un avvertimento finale: gli argomenti di `scanf` e di `sscanf` *devono* essere dei puntatori. L’errore più comune che si riscontra è

```
scanf(“%d”, n);
```

al posto di

```
scanf(“%d”, &n);
```

Nella maggior parte dei casi, questo errore non viene rilevato dal compilatore.

Esercizio 7.4 Scrivete una vostra versione di `scanf` analoga a `minprintf`, scritta nella sezione precedente.

Esercizio 7.5 Riscrivete la calcolatrice in notazione postfissa del Capitolo 4, usando `scanf` e/o `sscanf` per effettuare le conversioni dell’input e dei numeri.

7.5 Accesso a File

Gli esempi visti fino a questo momento leggevano dallo standard input e scrivevano sullo standard output, che sono entrambi definiti dal sistema operativo locale.

Il passo successivo consiste nello scrivere un programma che accede ad un file al quale *non* è già connesso. Un programma che evidenzia la necessità di una simile operazione è `cat`, che concatena un insieme di file sullo standard output. `cat` viene usato per stampare file sullo schermo, e come collettore dell’input per quei programmi che non sono in grado di accedere ai file tramite il loro nome. Per esempio, il comando

```
cat x.c y.c
```

stampa sullo standard output il contenuto dei file `x.c` e `y.c` (e nient'altro).

Il problema consiste nel fare in modo che i file nominati vengono letti, cioè nel connettere i nomi esterni pensati dall'utente alle istruzioni che leggono i dati.

Le regole sono semplici. Prima di poter essere letto o scritto, un file dev'essere *aperto* con la funzione di libreria `fopen`. `fopen` legge un nome esterno, come `x.c` o `y.c`, esegue alcune operazioni di carattere gestionale ed alcune chiamate al sistema operativo (i cui dettagli non ci interessano), e restituisce un puntato-re che dev'essere utilizzato nelle successive letture o scritture del file.

Questo puntatore, chiamato *file pointer*, punta ad una struttura che contiene informazioni sul file, come l'indirizzo di un buffer, la posizione corrente nel buffer, se il file è stato aperto in lettura o scrittura, e se si sono verificati errori oppure è stata raggiunta la fine del file. Gli utenti non hanno bisogno di conoscere questi dettagli, perché le definizioni contenute nell'header `<stdio.h>` comprendono la dichiarazione di una struttura chiamate `FILE`. L'unica dichiarazione necessaria per poter utilizzare un file pointer è esemplificata da

```
FILE  *fp;
FILE  *fopen(char *name, char *mode);
```

Questa dichiarazione afferma che `fp` è un puntatore a `FILE`, e che `fopen` restituisce un puntatore a `FILE`. Notate che `FILE` è un nome di tipo, come `int`, e non il tag di una struttura; esso è definito con una `typedef` (i dettagli dell'implementazione di `fopen` nel sistema operativo UNIX sono dati nella Sezione 8.5).

La chiamata a `fopen` in un programma è

```
fp=fopen(name, mode);
```

Il primo argomento di `fopen` è una stringa contenente il nome del file. Il secondo argomento è il *modo*, anch'esso una stringa, che indica come si intende utilizzare il file. I modi possibili sono la lettura ("`r`"), la scrittura ("`w`") e l'aggiunta o append ("`a`"). Alcuni sistemi distinguono tra file di testo e file binari; per questi ultimi, alla stringa della modalità deve essere aggiunto il suffisso "`b`".

Se un file che non esiste viene aperto in scrittura o in append, se è possibile esso viene creato. L'apertura in scrittura di un file già esistente fa sì che i vecchi contenuti vengano persi, diversamente da quando acca-de con l'apertura in modalità append. Tentare di leggere un file che non esiste è un errore, così come è un errore tentare di leggere un file sul quale non si ha permesso di accesso. Qualunque sia l'errore che si verifica, `fopen` ritorna `NULL` (l'errore può essere poi identificato con maggiore precisione; si veda la discussio-ne sulla gestione degli errori, al termine della Sezione 1 nell'Appendice B).

Una volta aperto il file, è necessario avere degli strumenti che consentano di leggerlo e scriverlo. Esistono diverse possibilità, le più semplici delle quali sono `getc` e `putc`. `getc` ritorna un carattere prelevato da un file; per poter identificare il file da cui leggere, essa deve conoscere il file pointer.

```
int  getc(FILE *fp)
```

`getc` ritorna il prossimo carattere dal file `fp`; ritorna EOF per errore o fine file.

```
int  putc(int c, FILE *fp)
```

`putc` scrive il carattere `c` sul file `fp` e ritorna il carattere scritto, oppure `EOF` se si verifica un errore. Come `getchar` e `putchar`, `getc` e `putc` possono essere delle macro, piuttosto che delle funzioni.

Quando un programma C entra in esecuzione, l'ambiente del sistema operativo si incarica di aprire tre file e di fornire i rispettivi puntatori. Questi file sono lo standard input, lo standard output e lo standard error; i puntatori corrispondenti sono chiamati `stdin`, `stdout` e `stderr`, dichiarati in `<stdio.h>`. In genere, `stdin` è associato alla tastiera, mentre `stdout` e `stderr` sono associati al video, ma `stdin` e `stdout` possono essere rediretti su altri file o pipe, secondo quanto descritto nella Sezione 7.1.

`getchar` e `putchar` possono essere definite in termini di `getc`, `putc`, `stdin` e `stdout` nel modo seguente:

```
#define  getchar()      getc(stdin)
#define  putchar(c)     putc((c), stdout)
```

Per l'input e l'output formattati sui file, si possono usare le funzioni `fscanf` e `fprintf`. Esse sono identi-che a `scanf` e `printf`, ad eccezione del fatto che il loro primo argomento è un file pointer che specifica il file da cui leggere o sul quale scrivere; la stringa di formato è il secondo argomento.

```
int  fscanf(FILE *fp, char *format, ...)
int  fprintf(FILE *fp, char *format, ...)
```

Una volta definiti questi aspetti preliminari, siamo in grado di scrivere il programma `cat`. Il suo schema è quello già utilizzato per altri programmi. Se alla linea di comando vengono forniti degli argomenti, essi vengono interpretati come nomi di file, ed esaminati nell'ordine in cui compaiono. In mancanza di argomenti, il file trattato è lo standard input.

```
#include  <stdio.h>

/* cat: concatena file, prima versione */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);

    if (argc==1)      /* non ci sono argomenti; copia lo standard input */
        filecopy(stdin, stdout);
    else
        while (--argc>0)
            if ((fp=fopen(*++argv, "r"))==NULL)
            {
                printf("Cat: non posso aprire %s\n", *argv);
                return 1;
            }
            else
            {
                filecopy(fp, stdout);
                fclose(fp);
            }
    return 0;
}

/* filecopy: copia il file ifp sul file ofp */
void  filecopy(FILE *ifp, FILE *ofp)
{
    int c;

    while ((c=getchar(ifp))!=EOF)
        putc(c, ofp);
}
```

I file pointer `stdin` e `stdout` sono oggetti di tipo `FILE *`. Tuttavia, essi sono delle costanti, *non* delle variabili, e quindi non è possibile assegnare loro oggetti diversi.

La funzione

```
int  fclose(FILE *fp)
```

è l'inverso di `fopen`; essa interrompe la connessione, creata da `fopen`, fra il puntatore al file ed il suo no-me esterno, rilasciando il puntatore per un altro file. Poiché molti sistemi operativi impongono dei limiti al numero di file che un programma può aprire contemporaneamente, è buona norma rilasciare i puntatori a file non appena essi diventano inutili, come abbiamo fatto nel programma `cat`. Esiste anche un'altra ragio-ne che giustifica l'esecuzione di `fclose` sul file di output: questa chiamata scrive effettivamente sul file il buffer in cui `putc` inserisce l'output. Quando un programma termina senza errori, `fclose` viene chiamata

automaticamente su tutti i file che esso aveva aperto (se non sono necessari, potete chiudere esplicitamen-te `stdin` e `stdout`, che possono anche venire riassegnati tramite la funzione di libreria `freopen`).

7.6 Gestioni degli Errori - STDERRED EXIT

Il trattamento degli errori realizzato nel programma `cat` non è il migliore possibile. Infatti se, per una ragio-ne qualsiasi, uno dei file non può essere aperto, il messaggio di errore viene stampato soltanto al termine dell'output concatenato. Questo può essere accettabile soltanto se l'output è diretto sullo schermo, ma non se è rediretto su un altro file o se costituisce l'input di un altro programma (tramite una pipe).

Per gestire meglio questa situazione, ad ogni programma viene assegnato un secondo flusso di output, det-to `stderr` (standard error). In genere, l'output scritto su `stderr` compare sul video anche se lo standard output è stato rediretto.

Rivediamo il programma `cat` in modo da fargli scrivere i messaggi di errore sullo standard error.

```
#include  <stdio.h>

/* cat: concatena file, seconda versione */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    char *prog=argv[0];   /* nome del programma; utile per gli errori */

    if (argc==1)          /* nessun argomento; copia lo standard input */
        filecopy(stdin, stdout);
    else
        while (--argc>0)
            if ((fp=fopen(*++argv, "r"))==NULL)
            {
                fprintf(stderr, "%s: non posso aprire %s\n", prog, *argv);
                exit(1);
            }
            else
            {
                filecopy(fp, stdout);
                fclose(fp);
            }
    if (ferror(stdout))
    {
        fprintf(stderr, "%s: errore in scrittura su stdout\n", prog);
        exit(2);
    }
    exit(0);
}
```

Il programma segnala gli errori in due modi. Da un lato, l'output diagnostico prodotto da `fprintf` va su `stderr`, in modo da potere raggiungere lo schermo, invece di scomparire in una pipe o in un file di output. Nel messaggio abbiamo incluso il nome del programma, `argv[0]`; in questo modo, se questo programma viene eseguito insieme ad altri, la fonte dell'errore è sempre identificabile.

In secondo luogo, il programma utilizza la funzione della libreria standard `exit`, che quando viene chiamata termina l'esecuzione di un programma. L'argomento di `exit` è a disposizione di qualunque processo abbia chiamato quello terminato da `exit` stessa.

Convezionalmente, un valore di ritorno nullo segnala una terminazione corretta; valori diversi da zero se-gnalano invece situazioni anomale. `exit` chiama `fclose` su ogni file di output aperto dal processo, per po-tere scaricare su file qualsiasi residuo di output bufferizzato.

All'interno di `main`, `return` *expr* equivale a `exit(`*expr*`)`. `exit` ha il vantaggio di poter essere invocata an-che da altre funzioni, e queste chiamate possono essere trovate con un programma di ricerca di un pattern come quello illustrato nel Capitolo 5.

La funzione `ferror` ritorna un valore non nullo se su `fp` si è verificato qualche errore.

```
int  ferror(FILE *fp)
```

Anche se piuttosto raramente, gli errori di output si verificano (per esempio, se su un disco non c'è più spazio), quindi i programmi dovrebbero sempre controllarne la presenza.

La funzione `feof(FILE *)` è analoga a `ferror`; essa ritorna un valore non nullo se è stata raggiunta la fine del file specificato

```
int  feof(FILE *fp)
```

Nei nostri brevi programmi illustrativi, non ci siamo in genere preoccupati di verificare gli stati di uscita, ma qualsiasi programma ben scritto dovrebbe gestire valori di ritorno significativi ed utili.

7.7 Input ed Output di Linee

La libreria standard fornisce una routine di input, `fgets`, simile alla funzione `getline` usata nei capitoli precedenti:

```
char  *fgets(char *line, int maxline, FILE *fp)
```

`fgets` legge una linea di input (compreso il new line) dal file `fp` e la registra nel vettore `line`; essa legge al più `maxline-1` caratteri. La linea risultante termina con uno `'\0'`. Normalmente `fgets` ritorna `line`; al termine del file, oppure alla rilevazione di un errore, essa ritorna invece `NULL` (la nostra funzione `getline` ritorna la lunghezza della linea, che è un valore più utile; zero, in questo caso, indica la fine del file).

Per l'output, la funzione `fputs` scrive una stringa (che può anche non contenere un new line) su un file:

```
int  fputs(char *line, FILE *fp)
```

Essa ritorna `EOF` se si verifica un errore, e zero altrimenti.

Le funzioni di libreria `gets` e `puts` sono simili a `fgets` e `fputs`, ma operano su `stdin` e `stdout`, rispettivamente. Notiamo che `gets` cancella il carattere `'\n'` finale, mentre `puts` lo aggiunge.

Per mostrare come le funzioni `fgets` e `fputs` non presentino alcuna caratteristica particolare, le presentiamo qui di seguito, così come appaiono nella libreria standard del nostro sistema:

```
/* fgets: preleva al più n caratteri da iop */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs=s;
    while (--n>0  &&  (c=getc(iop))!=EOF)
        if ((*cs++=c)=='\n')
            break;
    *cs='\0';
    return (c==EOF  &&  cs==s)?NULL:s;
}

/* fputs: scrive s sul file iop */
int  fputs(char *s, FILE *iop)
{
    int c;

    while (c=*s++)
        putc(c, iop);
    return ferror(iop)?EOF:0;
}
```

Per ragioni non ovvie, lo standard specifica valori di ritorno differenti per ferror e fputs.

È facile implementare la nostra funzione getline sfruttando fgets:

```
/* getline: legge una linea, ne ritorna la lunghezza */
int  getline(char *line, int max)
{
    if (fgets(line, max, stdin)==NULL)
        return 0;
    else
        return strlen(line);
}
```

Esercizio 7.6 Scrivete un programma per confrontare due file, che stampi la prima linea nella quale essi differiscono.

Esercizio 7.7 Modificate il programma di ricerca di un pattern scritto nel Capitolo 5, in modo che esso prelevi il suo input da un insieme di file elencati oppure, se non viene indicato alcun file come argomento, dallo standard input. Quando viene trovata una linea contenente il pattern voluto, è bene stampare il nome del file?

Esercizio 7.8 Scrivete un programma per stampare un insieme di file, iniziando la stampa di ognuno con una nuova pagina, e scrivendo un titolo ed un numero di pagina corrente per ogni file.

7.8 Funzioni Varie

La libreria standard fornisce un'ampia varietà di funzioni. Questa sezione è un elenco della sintassi delle più utili. Maggiori dettagli e molte altre funzioni si possono trovare nell'Appendice B.

7.8.1 Operazioni sulle Stringhe

Abbiamo già nominato le funzioni strlen, strcpy, strcat e strcmp, contenute in <string.h>. Nello elenco che segue, s e t sono di tipo char *, mentre c e n sono di tipo int.

strcat(s, t)	concatena t alla fine di s
strncat(s, t, n)	concatena n caratteri di t alla fine di s
strcmp(s, t)	ritorna un valore negativo, nullo o positivo per s<t, s==t, s>t
strncmp(s, t, n)	uguale a strcmp, ma confronta solo i primi n caratteri
strcpy(s, t)	copia t in s
strncpy(s, t, n)	copia al più n caratteri di t in s
strlen(s)	ritorna la lunghezza di s
strchr(s, c)	ritorna un puntatore al primo c in s, NULL se c non compare in s
strrchr(s, c)	ritorna un puntatore all'ultimo c in s, NULL se c non compare in s

7.8.2 Controllo e Conversione della Classe di un Carattere

Alcune funzioni, definite in <ctype.h>, eseguono controlli e conversioni sui caratteri. Nell'elenco che segue, c è un int che può essere rappresentato come un unsigned char, oppure EOF. Le funzioni ritornano valori int.

isalpha(c)	diverso da zero se c è alfabetico, 0 altrimenti
isupper(c)	diverso da zero se c è maiuscolo, 0 altrimenti
islower(c)	diverso da zero se c è minuscolo, 0 altrimenti
isdigit(c)	diverso da zero se c è una cifra, 0 altrimenti
isalnum(c)	diverso da zero se è vero isalpha(c) o isdigit(c), 0 altrimenti
isspace(c)	diverso da zero se c è un carattere di spaziatura, 0 altrimenti
toupper(c)	ritorna c convertito in maiuscolo
tolower(c)	ritorna c convertito in minuscolo

7.8.3 UNGETC

La libreria standard fornisce una versione piuttosto ristretta della funzione ungetch che abbiamo scritto nel Capitolo 4; tale versione si chiama ungetc.

```
int  ungetc(int c, FILE *fp)
```

rideposita il carattere c nel file fp, e restituisce c oppure EOF, se rileva un errore. Per ogni file viene garantito il deposito di un solo carattere. ungetc può essere utilizzata con una qualsiasi delle funzioni di input scanf, getc o getchar.

7.8.4 Esecuzione di Comandi

La funzione system(char *s) esegue il comando contenuto nella stringa s, ed in seguito riprende l'esecuzione del programma corrente. Il contenuto di s dipende fortemente dal sistema operativo locale. Come esempio banale, sui sistemi UNIX, l'istruzione

```
system("date");
```

provoca l'esecuzione del programma date; esso stampa la data e l'ora del giorno sullo standard output. system ritorna un intero, dipendente dal sistema, relativo allo stato di terminazione del programma eseguito. Nei sistemi UNIX, lo stato di ritorno è il valore restituito da exit.

7.8.5 Gestione della Memoria

Le funzioni malloc e calloc allocano dinamicamente blocchi di memoria.

```
void  *malloc(size_t n)
```

ritorna un puntatore a n byte di memoria non inizializzata, oppure NULL se la richiesta non può essere soddisfatta.

```
void  *calloc(size_t n, size_t size)
```

ritorna un puntatore ad un'area sufficiente a contenere un vettore di n oggetti dell'ampiezza specificata, oppure NULL se la richiesta non può essere soddisfatta. La memoria è inizializzata a zero. Il puntatore restitui-to da malloc e calloc ha l'allineamento corretto per gli oggetti in questione, ma deve essere forzato dal tipo appropriato, come in

```
int *ip;
ip=(int *) calloc(n, sizeof(int));
```

free(p) libera lo spazio puntato da p, dove p è un puntatore ottenuto tramite una precedente chiamata a malloc o calloc. Non esiste alcuna restrizione sull'ordine in cui lo spazio può essere liberato, ma è comunque un errore rilasciare spazio che non è stato ottenuto tramite malloc o calloc.

Anche usare un'area già rilasciata in precedenza è un errore. Un segmento di codice molto frequente ma scorretto è questo ciclo, che libera elementi di una lista:

```
for (p=head; p!=NULL; p=p->next)          /* SBAGLIATO */
    free(p);
```

Il modo corretto di eseguire quest'operazione consiste nel salvare tutto ciò che serve prima della free:

```
for (p=head; p!=NULL; p=q)
{
    q=p->next;
    free(p);
}
```

La Sezione 8.7 illustra l'implementazione di un allocatore come malloc, nel quale i blocchi allocati possono essere rilasciati in un ordine qualsiasi.

7.8.6 Funzioni Matematiche

Esistono più di venti funzioni matematiche, dichiarate in <math.h>; l'elenco che segue comprende quelle maggiormente utilizzate. Ognuna di esse ha uno o più argomenti di tipo `double`, e ritorna a sua volta un `double`.

sin(*x*)	seno di *x*, con *x* espresso in radianti
cos(*x*)	coseno di *x*, con *x* espresso in radianti
atan2(*y*,*x*)	arcotangente di *y/x*, in radianti
exp(*x*)	funzione esponenziale e^x
log(*x*)	logaritmo naturale (base *e*) di *x* ($x > 0$)
log10(*x*)	logaritmo comune (base 10) di *x* ($x > 0$)
pow(*x*,*y*)	x^y
sqrt(*x*)	radice quadrata di *x* ($x >= 0$)
fabs(*x*)	valore assoluto di *x*

7.8.7 Generazione di Numeri Casuali

La funzione `rand()` calcola una sequenza di interi pseudo-casuali nell'intervallo `0`, `RAND_MAX`, che è definito in <stdlib.h>. Un modo per produrre numeri casuali floating-point maggiori o uguali a zero ma minori di uno è

```
#define  frand()   ((double)rand()/(RAND_MAX+1))
```

Se la vostra libreria fornisce già una funzione per i numeri casuali in floating-point, essa avrà, probabilmen-te, proprietà statistiche migliori di questa.

La funzione `srand(unsigned)` stabilisce il seme per `rand`. L'implementazione portabile di `rand` e `srand` suggerita dallo standard è quella presentata nella Sezione 2.7.

Esercizio 7.9 Funzioni come `isupper` possono essere implementate per risparmiare spazio e tempo. Esplorate entrambe queste possibilità.

CAPITOLO 8

L'INTERFACCIA DEL SISTEMA UNIX

Il sistema operativo UNIX fornisce i suoi servizi attraverso un insieme di *chiamate di sistema* che sono, in ultima analisi, delle funzioni interne al sistema operativo stesso e che possono essere invocate dai programmi utente. Questo capitolo descrive come usare nei programmi C alcune fra le più importanti chiamate di sistema. Se utilizzate un sistema UNIX, questo capitolo vi sarà di immediata utilità, poiché spesso le chiamate di sistema sono indispensabili per il raggiungimento dell'efficienza massima, o per utilizzare funzionalità non comprese nella libreria. Tuttavia, anche se usate il C su un sistema operativo diverso, lo studio dei prossimi esempi dovrebbe consentirvi di approfondire notevolmente la vostra conoscenza del linguaggio; infatti, anche se i dettagli variano da sistema a sistema, le diverse implementazioni sono comunque sempre piuttosto simili. Poiché la libreria ANSI C è spesso modellata sulle funzionalità di UNIX, questo codice può aiutarvi, fra l'altro, a comprendere meglio la libreria stessa.

Il capitolo è suddiviso in tre parti principali: l'input / output, il file system e l'allocazione della memoria. Le prime due sezioni assumono che il lettore abbia una seppur limitata familiarità con le caratteristiche esterne dei sistemi UNIX.

Il Capitolo 7 si riferiva ad un'interfaccia di input / output uniforme sui vari sistemi operativi. Su un particolare sistema operativo, le routine della libreria standard devono essere però implementate utilizzando le funzio-nalità disponibili sul sistema in questione. Nelle prossime sezioni descriveremo le chiamate del sistema UNIX per l'input e l'output, e mostreremo come esse sono state impiegate nella realizzazione di alcune parti della libreria standard.

8.1 Descrittori di File

Nel sistema operativo UNIX, tutto l'input / output viene effettuato leggendo o scrivendo file, perché tutte le periferiche, compresi la tastiera ed il video, sono considerati dei file all'interno del file system. Questo signi-fica che la gestione della comunicazione fra un programma e le periferiche è lasciata ad un'unica interfaccia omogenea.

Nel caso più generale, prima di leggere o scrivere un file dovete informare il sistema delle vostre intenzioni, *aprendo* il file stesso. Nel caso in cui vogliate scrivere sul file, è anche necessario crearlo o, se esiste già, cancellarne il contenuto. Il sistema controlla che voi abbiate il permesso di fare queste operazioni e, in caso affermativo, restituisce al programma un piccolo intero non negativo, chiamato *descrittore di file*. Ogni volta che dev'essere eseguito dell'input / output sul file, quest'ultimo viene identificato attraverso il suo descrittore, invece che tramite il nome (un descrittore di file è analogo al file pointer usato nella libreria standard). Tutte le informazioni relative ad un file aperto vengono gestite dal sistema: il programma utente accede al file soltanto attraverso il descrittore.

Poiché le operazioni di input / output che coinvolgono la tastiera ed il video sono molto frequenti, per questi dispositivi esistono accorgimenti particolari. Quando l'interprete di comandi (la "shell") esegue un program-ma, vengono aperti tre file aventi descrittori 0, 1 e 2, detti rispettivamente standard input, standard output e standard error. Se un programma legge dal descrittore 0 e scrive sui descrittori 1 e 2, esso esegue dell'input / output senza dover aprire esplicitamente alcun file.

L'utente di un programma può redirigere l'I/O da e su file usando < e >:

```
prog  <infile  >outfile
```

In questo caso, la shell modifica gli assegnamenti di default dei descrittori 0 e 1, associandoli ai file nominati nella linea di comando. In genere, il descrittore 2 rimane associato allo schermo, in modo che i messaggi di errore possano essere visualizzati. Osservazioni molto simili alle precedenti valgono anche per l'input / output associato ad una pipe. In ogni caso, gli assegnamenti dei file sono a carico della shell, e non del programma utente, che non sa né da dove proviene il suo input né dove è diretto il suo output, poiché si limita ad utilizzare i descrittori 0 per il primo e 1 e 2 per il secondo.

8.2 I/O a Basso Livello - READ e WRITE

L'input / output sfrutta le chiamate di sistema `read` e `write`, alle quali un programma C accede attraverso due funzioni dette, a loro volta, `read` e `write`. Per entrambe, il primo argomento è un descrittore di file. Il secondo argomento è un vettore di caratteri, appartenente al programma, nel quale i dati vengono registrati o dal quale provengono. Il terzo argomento è il numero dei byte da trasferire.

```
int  n_read=read(int fd, char *buf, int n);
int  n_written=write(int fd, char *buf, int n);
```

Ogni chiamata restituisce il numero di byte trasferiti. In lettura, tale numero può essere inferiore a quello ri-chiesto. Un valore di ritorno nullo significa che è stata raggiunta la fine del file, mentre in caso di errore viene fornito il valore `-1`. In scrittura, il valore di ritorno è il numero di byte scritti; in questo caso, il numero fornito differisce da quello richiesto soltanto in caso di errore.

In un'unica chiamata è possibile trasferire un qualsiasi numero di byte. I valori più comuni sono 1, se si vuole leggere un carattere per volta (operazione "non bufferizzata"), oppure un numero come 1024 o 4096, che corrisponde alla dimensione di un blocco fisico sulla periferica. Valori maggiori possono aumentare l'efficienza, perché consentono di effettuare un numero inferiore di chiamate di sistema.

Consideriamo tutti questi fattori, siamo in grado di scrivere un semplice programma che copia il suo input nel suo output, equivalente al programma per la copia di file scritto nel Capitolo 1. Questo programma copierà qualsiasi altra, poiché l'input l'output possono essere rediretti su qualunque file o device.

```
#include  "syscalls.h"

main()      /* copia l'input sull'output */
{
    char buf[BUFSIZ];
    int n;

    while ((n=read(0, buf, BUFSIZ))>0)
        write(1, buf, n);
    return 0;
}
```

Abbiamo riunito i prototipi di funzione delle chiamate di sistema in un unico file chiamato `syscalls.h`, che verrà incluso dalle funzioni di questo capitolo. Notiamo, tuttavia, che questo nome non è standard.

Anche il parametro `BUFSIZ` è definito in `syscalls.h`; il suo valore corrisponde ad un'ampiezza particolar-mente adatta al sistema sul quale si opera. Se la dimensione del file non è multipla di `BUFSIZ`, una `read` ri-tornerà un numero di byte inferiore a `BUFSIZ`, e corrispondente al numero di byte da scrivere; la chiamata successiva a `read` ritornerà zero.

È istruttivo vedere come `read` e `write` possano essere utilizzate per costruire routine di più alto livello, co-me `getchar` e `putchar`. Per esempio, illustriamo una versione di `getchar` che effettua dell'input non buf-ferizzato, leggendo dallo standard input un carattere per volta.

```
#include  "syscalls.h"

/* getchar: input non bufferizzato di un singolo carattere */
int  getchar(void)
{
    char c;

    return (read(0, &c, 1)==1)?(unsigned char)c:EOF;
}
```

`c` dev'essere di tipo `char`, perché `read` vuole un puntatore a carattere. Forzare `c` ad un `unsigned char`, nell'istruzione `return`, elimina qualsiasi problema di estensione del segno.

La seconda versione di `getchar` esegue l'input a blocchi di grandi dimensioni, e rilascia ad uno ad uno i caratteri verso l'esterno.

```
#include  "syscalls.h"
```

```
/* getchar: versione semplice bufferizzata */
int  getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp=buf;
    static int n=0;

    if (n==0)           /* il buffer è vuoto */
    {
        n=read(0, buf, sizeof buf);
        bufp=buf;
    }
    return (--n>=0)?(unsigned char)*bufp++:EOF;
}
```

Se queste versioni di `getchar` fossero state compilate con `<stdio.h>`, sarebbe stato necessario inserire la linea `#undef  getchar`, per gestire la possibilità che `getchar` fosse già definita come macro.

8.3 OPEN, CREAT, CLOSE, UNLINK

Per leggere o scrivere file diversi dagli standard input, output ed error di default, è necessario aprirli esplici-tamente. Per poterlo fare, il C fornisce due chiamate di sistema: `open` e `creat`.

`open` è molto simile alla funzione `fopen` discussa nel Capitolo 7, eccetto che per il fatto che essa restituisce un descrittore di file, cioè un `int`, invece di un file pointer, come `fopen`. In caso di errore, `open` restituisce `-1`.

```
#include  <fcntl.h>

int fd;
int open(char *name, int flags, int perms);

fd=open(name, flags, perms);
```

Come per `fopen`, l'argomento `name` è una stringa contenente il nome del file. Il secondo argomento, `flags`, è un `int` che specifica come aprire il file; i suoi valori più frequenti sono:

```
O_RDONLY          apre solo in lettura
O_WRONLY          apre solo in scrittura
O_RDWR            apre sia in lettura che in scrittura
```

Queste costanti sono definite in `<fcntl.h>` su sistemi UNIX System V, ed in `<sys/file.h>` nelle versioni di Berkeley (BSD).

Per aprire in lettura un file già esistente, scriviamo

```
fd=open(name, O_RDONLY, 0);
```

L'argomento `perms`, negli esempi di `open` che vedremo, sarà sempre nullo.

È un errore tentare di aprire un file che non esiste. Per creare nuovi file, oppure per riscrivere quelli già esi-stenti, bisogna utilizzare la chiamata di sistema `creat`:

```
int  creat(char *name, int perms);

fd=creat(name, perms);
```

fornisce un descrittore di file, se riesce a creare il file richiesto, e `-1` altrimenti. Se il file esiste già, `creat` tronca a zero la sua lunghezza, perdendo quindi il vecchio contenuto; non è un errore creare un file esiste già.

Se il file non esiste, `creat` lo crea con i permessi specificati dall'argomento `perms`. Nel file system UNIX, ad ogni file sono associati nove bit che controllano i permessi di lettura, scrittura ed esecuzione del file da parte

del suo proprietario, del gruppo al quale egli appartiene e di tutti gli altri utenti del sistema. Di conseguenza, i permessi di accesso possono essere convenientemente specificati da un numero ottale. Per esempio, `0755` specifica che il file può essere letto, scritto ed eseguito dal suo proprietario, mentre può essere soltanto letto e scritto dal suo gruppo e da tutti gli altri utenti.

Per illustrare questi concetti, presentiamo nel seguito una versione semplificata del programma UNIX `cp`, che copia un file in un altro. La nostra versione copia soltanto un file, non accetta come secondo argomento una directory e, invece di copiarli dal file di partenza, inventa i permessi di accesso.

```
#include  <stdio.h>
#include  <fcntl.h>
#include  “syscalls.h”
#define  PERMS  0666   /* RW per proprietario, gruppo e altri */

void  error(char *, ...);

/* cp: copia f1 in f2 */
main(int argc, char *argv[])
{
    int f1, f2, n;
    char buf[BUFSIZ];

    if (argc!=3)
        error(“Utilizzo: cp da a”);
    if ((f1=open(argv[1], O_RDONLY, 0))==-1)
        error(“cp: non posso aprire %s”, argv[1]);
    if ((f2=creat(argv[2], PERMS))==-1)
        error(“cp: non posso creare %s, modo %03o”, argv[2], PERMS);
    while ((n=read(f1, buf, BUFSIZ))>0)
        if (write(f2, buf, n)!=n)
            error(“cp: errore di scrittura sul file %s”, argv[2]);
    return 0;
}
```

Questo programma crea il file di output con dei permessi di accesso fissati a `0666`. Usando la chiamata di sistema `stat`, descritta nella Sezione 8.6, è possibile determinare i permessi di un file già esistente ed assegnarli al risultato della copia.

Notate che la funzione `error` viene invocata con un numero variabile di argomenti, come `printf`. L’imple-mentazione di `error` illustra come utilizzare un altro membro della famiglia di `printf`. La funzione della li-breria standard `vprintf` è uguale a `printf`, ad eccezione del fatto che in essa la lista variabile di argomenti è sostituita da un singolo argomento, precedentemente inizializzato con una chiamato alla macro `va_start`. Analogamente, `vfprintf` e `vsprintf` sono le versioni corrispondenti a `fprintf` e `sprintf`.

```
#include  <stdio.h>
#include  <stdarg.h>

/* error: stampa un messaggio di errore e conclude il programma */
void  error(char *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    fprintf(stderr, “errore: ”);
    vprintf(stderr, fmt, args);
    fprintf(stderr, “\n”);
    va_end(args);
    exit(1);
}
```

Esiste un limite (che in genere si aggira intorno a 20) al numero di file che un programma può tenere aperti contemporaneamente. Di conseguenza, un programma che intende usare molti file deve predisporsi a riuti-lizzare più volte gli stessi descrittori. La funzione `close(int fd)` spezza la connessione esistente tra un descrittore di file ed un file aperto, e libera il descrittore, in modo che sia disponibile per un altro file; questa funzione corrisponde alla funzione `fclose` della libreria standard, con la differenza che la prima non

svuota i buffer. La terminazione di un programma tramite exit o tramite l'uscita dal main provoca la chiusura au-tomatica di tutti i file aperti.

La funzione unlink(char *name) rimuove il file name dal file system. Essa corrisponde alla funzione re-move della libreria standard.

Esercizio 8.1 Riscrivete il programma cat del Capitolo 7 usando read, write, open e close invece dei loro equivalenti della libreria standard. Effettuate delle prove per determinare la velocità relativa delle due versioni.

8.4 Accesso Casuale – LSEEK

Normalmente, l'input e l'output sono sequenziali: ogni read o write opera nella posizione del file immedia-tamente successiva a quella in cui si è svolta l'operazione precedente. Tuttavia, quando è necessario, un fi-le può essere letto o scritto in un ordine arbitrario. La chiamata di sistema lseek consente lo spostamento sul file senza che avvenga alcuna lettura o scrittura di dati:

```
long lseek(int fd, long offset, int origin);
```

inizializza la posizione corrente del file con descrittore fd a offset, che viene considerato come un valore relativo rispetto a origin. Le successive letture o scritture inizieranno a quella posizione. origin può va-lere 0, 1 0 2, ed indica che offset dev'essere calcolato, rispettivamente, dall'origine del file, dalla posizio-ne corrente o dalla fine. Per esempio, per appendere dati ad un file (con l'operatore >> di redirezione della shell di UNIX, o con l'argomento "a" per fopen), è necessario posizionarsi in scrittura al termine del file:

```
lseek(fd, 0L, 2);
```

Per tornare all'inizio ("rewind"),

```
lseek(fd, 0L, 0);
```

Osservate l'argomento 0L; avremmo anche potuto scriverlo come (long)0, o semplicemente 0, se lseek fosse stata dichiarata nel modo opportuno.

Usando lseek è possibile trattare i file quasi come i vettori, pagando però un maggiore costo di accesso. Per esempio, la funzione che segue legge un qualsiasi numero di byte da una posizione arbitraria del file. Essa ritorna il numero letto, oppure -1 in caso di errore.

```
#include  "syscalls.h"

/* get: legge n byte dalla posizione pos */
int  get(int fd, long pos, char *buf, int n)
{
    if (lseek(fd, pos, 0)>=0)       /* si posiziona su pos */
        return read(fd, buf, n);
    else
        return -1;
}
```

Il valore restituito da lseek è un long che fornisce la nuova posizione del file, oppure -1 se si è verificato un errore. La funzione fseek, della libreria standard, è simile a lseek, ad eccezione del fatto che il suo pri-mo argomento è di tipo FILE * e che si limita a ritornare un valore diverso da zero in caso di errore.

8.5 Esempio – Un'Implementazione di FOPEN e GETC

Per illustrare come i concetti sin qui discussi siano collegati, mostriamo un'implementazione delle funzioni fopen e getc, fornite dalla libreria standard.

Ricordiamo che, nella libreria standard, i file sono descritti da file pointer, e non da descrittori. Un file pointer è un puntatore ad una scrittura che contiene informazioni relative al file: un puntatore ad un buffer, che con-sente di leggere il file a blocchi di grandi dimensioni; un contatore del numero di caratteri rimasti nel

buffer; un puntatore alla posizione del successivo carattere nel buffer; il descrittore del file; alcuni flag che descri-vono la modalità di apertura (lettura / scrittura), lo stato di errore ecc.

La struttura che descrive un file si trova in `<stdio.h>`, che dev'essere incluso (tramite `#include`) da tutti i file sorgente che utilizzano routine di input / output della libreria standard. Questo stesso file è incluso anche dalle funzioni della libreria stessa. Nel seguente estratto di un tipico `<stdio.h>`, i nomi usati soltanto da funzioni della libreria iniziano con il carattere underscore, il che li rende difficilmente confondibili con quelli contenuti nei programmi utente. Questa convenzione, notiamo, è comune a tutte le routine della libreria standard.

```
#define  NULL     0
#define  EOF      (-1)
#define  BUFSIZ   1024
#define  OPEN_MAX 20    /* massimo numeri di file aperti insieme */

typedef struct _iobuf {
    int cnt;            /* caratteri rimasti */
    char *ptr;          /* posizione del prossimo carattere */
    char *base;         /* indirizzo del buffer */
    int flag;           /* modalità di accesso al file */
    int fd;             /* descrittore di file */
} FILE;
extern FILE  _iob[OPEN_MAX];

#define  stdin   (&_iob[0])
#define  stdout  (&_iob[1])
#define  stderr  (&_iob[2])

enum _flags {
    _READ=01,           /* file aperto in lettura */
    _WRITE=02,          /* file aperto in scrittura */
    _UNBUF=04,          /* file non bufferizzato */
    _EOF=010,           /* EOF raggiunto sul file */
    _ERR=020            /* rilevamento di un errore */
};

int  _fillbuf(FILE *);
int  _flushbuf(int, FILE *);

#define  feof(p)    (((p)->flag & _EOF)!=0)
#define  ferror(p)  (((p)->flag & _ERR)!=0)
#define  fileno(p)  (((p)->fd)

#define  getc(p)    (--(p)->cnt>=0?(unsigned char)*(p)->ptr++:_fillbuf(p))
#define  putc(x, p) (--(p)->cnt>=0?*(p)->ptr++=(x):_flushbuf((x),p))

#define  getchar()   getc(stdin)
#define  putchar(x)  putc((x), stdout)
```

Normalmente la macro `getc` decrementa il contatore, sposta in avanti il puntatore e ritorna il carattere (ri-cordiamo che una `#define` molto lunga può proseguire su più righe, purché ognuna di queste, ad eccezio-ne dell'ultima, termini con il carattere backslash). Se però il contatore diventa negativo, `getc` chiama la fun-zione `_fillbuf` che riempie nuovamente il buffer, reinizializza il contenuto della struttura e restituisce un carattere. I caratteri vengono forniti come `unsigned`, il che assicura che siano tutti positivi.

Anche se non discuteremo tutti i dettagli, abbiamo incluso anche la definizione di `putc`, per mostrare come essa operi in modo analogo a `getc`, chiamando una funzione `_flushbuf` quando il buffer è pieno. Abbia-mo infine incluso anche le macro per il riconoscimento della fine del file, di uno stato di errore e del descrit-tore del file.

Ora possiamo scrivere la funzione `fopen`, buona parte della quale riguarda l'apertura ed il posizionamento sul file, oltre che l'inizializzazione dei bit di stato. `fopen` non effettua alcuna allocazione di spazio su un buf-fer; quest'operazione, infatti, viene svolta dalla routine `_fillbuf` al momento della prima lettura sul file.

```
#include  <fcntl.h>
#include  "syscalls.h"
#define  PERMS  0666    /* RW per proprietario, gruppo e altri */

/* fopen: apre file, restituisce un file pointer */
FILE  *fopen(char *name, char *mode)
{
    int fd;
    FILE  *fp;

    if (*mode!='r'  &&  *mode!='w'  &&  *mode!='a')
        return NULL;
    for (fp=_iob; fp<_iob+OPEN_MAX; fp++)
        if ((fp->flag & (_READ | _WRITE))==0)
            break;           /* ha trovato una posizione libera */
    if (fp>=_iob+OPEN_MAX)    /* non ci sono posizioni libere */
        return NULL;

    if (*mode=='w')
        fd=creat(name, PERMS);
    else if (*mode=='a')
    {
        if ((fd=open(name, O_WRONLY, 0))==-1)
            fd=creat(name, PERMS);
        lseek(fd, 0L, 2);
    }
    else
        fd=open(name, O_RDONLY, 0);
    if (fd==-1)               /* l'accesso a name non è consentito */
        return NULL;
    fp->fd=fd;
    fp->cnt=0;
    fp->base=NULL;
    fp->flag=(*mode=='r')?_READ:_WRITE;
    return fp;
}
```

Questa versione di `fopen` non gestisce tutte le modalità di accesso che sono fornite dallo standard, che po-trebbero comunque essere aggiunte senza l'inserimento di grandi quantità di codice. In particolare, la nostra `fopen` non riconosce la modalità "`b`", che segnala l'accesso binario, poiché sui sistemi UNIX essa non è si-gnificativa; neppure la modalità "`+`", per aprire in lettura e scrittura, è supportata da questa versione.

La prima chiamata a `getc` su un particolare file trova il contatore azzerato, e questo provoca una chiamata a `_fillbuf`. Se `_fillbuf` trova che il file non è aperto in lettura, essa restituisce immediatamente il ca-rattere `EOF`. Altrimenti, essa cerca di allocare un buffer (se la lettura è bufferizzata).

Una volta allocato il buffer, `_fillbuf` invoca, per riempirlo, la funzione `read`, inizializza il contatore ed i puntatori e restituisce il carattere che si trova all'inizio del buffer. Le chiamate successive a `_fillbuf` tro-veranno un buffer già allocato.

```
#include  "syscalls.h"

/* _fillbuf: alloca e riempie un buffer di input */
int  _fillbuf(FILE *fp)
{
    int bufsize;

    if ((fp->flag & (_READ | _EOF | _ERR))!=_READ)
        return EOF;
    bufsize=(fp->flag & _UNBUF)?1:BUFSIZ;
    if (fp->base==NULL)       /* non c'è ancora il buffer */
        if ((fp->base=(char *)malloc(bufsize))==NULL)
            return EOF;       /* non può allocare il buffer */
    fp->ptr=fp->base;
    fp->cnt=read(fp->fd, fp->ptr, bufsize);
    if (--fp->cnt<0)
```

```
    {
        if (fp->cnt==-1)
            fp->flag!=_EOF;
        else
            fp->flag!=_ERR;
        fp->cnt=0;
        return EOF;
    }
    return (unsigned char)*fp->ptr++;
}
```

L'unico punto ancora oscuro è il modo in cui tutto questo meccanismo viene innescato. Il vettore `_iob` deve essere definito ed inizializzato per `stdin`, `stdout` e `stderr`:

```
FILE _iob[OPEN_MAX]={          /* stdin, stdout, stderr */
    { 0, (char *) 0, _READ, 0},
    { 0, (char *) 0, _WRITE, 1},
    { 0, (char *) 0, _WRITE | _UNBUF, 2}
};
```

L'inizializzatore del campo `flag` della struttura mostra che `stdin` dev'essere letto, `stdout` dev'essere scritto, e `stderr` dev'essere scritto in modo non bufferizzato.

Esercizio 8.2 Riscrivete `fopen` e `_fillbuf` usando dei field invece che delle operazioni esplicite sui bit. Confrontate la dimensione del codice e la velocità delle sue versioni.

Esercizio 8.3 Progettate e scrivete `_flushbuf`, `fflush` e `fclose`.

Esercizio 8.4 La funzione della libreria standard

```
int fseek(FILE *fp, long offset, int origin)
```

è uguale a `lseek`, ad eccezione del fatto che `fp` è un file pointer invece di un descrittore di file, e che il va-lore restituito non è una posizione, bensì uno stato di tipo `int`. Scrivete `fseek`. Assicuratevi che la vostra versione coordini nel modo opportuno le bufferizzazioni effettuate dalle altre funzioni della libreria.

8.6 Esempio – Listing di Directory

Spesso è necessario interagire con il file system non per conoscere il contenuto di un file, bensì per ottene-re informazioni *sul* file. Un esempio di ciò è dato dal programma di listing di una directory, `ls`, fornito da UNIX: questo comando stampa i nomi dei file contenuti in una directory e, opzionalmente, altre informazioni quali le loro dimensioni, i permessi e così via. `ls` è, per UNIX, ciò che il comando `dir` è per MS-DOS.

Poiché in UNIX una directory non è altro che un file, per potere leggere i nomi dei file `ls` deve semplice-mente leggere la directory. Tuttavia alcune informazioni, quali la dimensione dei file, possono essere otte-nute soltanto attraverso delle chiamate di sistema. Su alcuni altri sistemi, come per esempio MS-DOS, an-che i nomi dei file presenti nella directory sono accessibili soltanto tramite chiamate di sistema. Il nostro in-tento è quello di fornire un accesso alle informazioni che sia indipendente dal sistema usato, anche se l'im-plementazione può dipendere fortemente da quest'ultimo.

Illustreremo alcuni degli aspetti legati a questo problema scrivendo un programma chiamato `fsize`. `fsize` è una versione particolare di `ls`, che stampa la dimensione di tutti i file forniti nella lista di argomenti alla li-nea di comando. Se uno dei file è una directory, `fsize` vi si applica ricorsivamente. Se non vengono forniti argomenti, `fsize` tratta la directory corrente.

Iniziamo con una breve revisione della struttura del file system di UNIX. Una *directory* è un file che contiene una lista di nomi di file, ed alcune indicazioni sulla loro posizione. Tale "posizione" è un indice in una tabella detta "lista degli inode". L'*inode* di un file è il luogo in cui vengono mantenute tutte le informazioni relative ad esso, ad eccezione del suo nome. Nella maggior parte dei casi, ogni entry della directory è costituita da un nome di file e dal numero dell'inode.

Purtroppo, il formato ed i contenuti di una directory non sono gli stessi su tutte le versioni del sistema. Per-ciò, cercheremo di suddividere il nostro problema in due segmenti, nel tentativo di identificare le parti non portabili. Il livello più esterno definisce una struttura chiamata `Dirent` e tre routine (`opendir`, `readdir` e `closedir`), che consentono di accedere al nome ed all'inode contenuti nella directory in modo indipenden-te dal tipo di file system. Scriveremo `fsize` usando quest'interfaccia. In seguito, illustreremo come imple-mentare queste routine su sistemi che usano directory con struttura analoga a quella adottata sulla Version 7 e su UNIX System V; le varianti sono lasciate come esercizio.

La struttura `Dirent` contiene il numero dell'inode ed il nome del file. La lunghezza massima di quest'ultimo è `NAME_MAX`, un valore dipendente dal sistema. `opendir` ritorna un puntatore ad una struttura chiamata `DIR`, analoga a `FILE`, usata da `readdir` e `closedir`. Quest'informazione è memorizzata in un file chia-mato `dirent.h`.

```
#define  NAME_MAX  14          /* massima lunghezza del nome;
                                  dipende dal sistema */

typedef struct {               /* entry della directory (portabile) */
    long ino;                  /* numero dell'inode */
    char name[NAME_MAX+1];     /* nome + '\0' */
} Dirent;

typedef struct {               /* DIR minima: niente bufferizzazione, */
    int fd;                    /* descrittore di file per la directory */
    Dirent d;                  /* entry della directory */
} DIR;

DIR  *opendir(char *dirname);
Dirent  *readdir(DIR *dfd);
void  closedir(DIR *dfd);
```

La chiamata di sistema `stat` prende in input un nome di file e ritorna tutte le informazioni contenute nel suo inode, oppure `-1` se si verifica un errore. Quindi,

```
char *name;
struct stat stbuf;
int  stat(char *, struct stat *);

stat(name, &stbuf);
```

inserisce nella struttura `stbuf` le informazioni contenute nell'inode relativo al file `name`. La struttura che de-scrive il valore ritornato da `stat` si trova in `<sys/stat.h>`, ed ha il seguente formato:

```
struct stat            /* informazioni dell'inode restituite da stat */
{
    dev_t  st_dev;     /* device dell'inode */
    ino_t  st_ino;     /* numero dell'inode */
    short  st_mode;    /* bit di modalità */
    short  st_nlink;   /* numero di link al file */
    short  st_uid;     /* user id del proprietario */
    short  st_gid;     /* group id del proprietario */
    dev_t  st_rdev;    /* per file speciali */
    off_t  st_size;    /* dimensione del file in caratteri */
    time_t st_atime;   /* data di ultimo accesso */
    time_t st_mtime;   /* data di ultima modifica */
    time_t st_ctime;   /* data di creazione */
};
```

Il significato di molti di questi campi è chiarito nei commenti. I tipi come `dev_t` e `ino_t` sono definiti in `<sys/types.h>`, che dev'essere incluso.

Il campo `st_mode` contiene un insieme di flag che descrivono il file. La definizione di questi flag si trova an-ch'essa in `<sys/stat.h>`; per i nostri scopi, è rilevante soltanto la parte di definizioni relative al tipo di file:

```
#define  S_IFMT    0160000         /* tipo del file */
#define  S_IFDIR   0040000         /* directory */
#define  S_IFCHR   0020000         /* speciale a caratteri */
#define  S_IFBLK   0060000         /* speciale a blocchi */
#define  S_IFREG   0100000         /* regolare */
```

Ora siamo in grado di scrivere il programma `fsize`. Se il modo ottenuto tramite `stat` indica che il file non è una directory, allora la sua dimensione è nota e può venire stampata direttamente. Se il file è una directo-ry, invece, dobbiamo analizzarla al ritmo di un file per volta; a sua volta, questa directory può contenere delle sotto-directory, e questo rende ricorsivo il processo di listing.

La routine principale gestisce gli argomenti della linea di comando; ogni argomento viene passato alla fun-zione `fsize`.

```
#include  <stdio.h>
#include  <string.h>
#include  "syscalls.h"
#include  <fcntl.h>           /* flag per lettura e scrittura */
#include  <sys/types.h>       /* definizioni di tipi */
#include  <sys/stat.h>        /* struttura ritornata da stat */
#include  "dirent.h"

void  fsize(char *);

/* stampa le dimensioni dei file */
main(int argc, char **argv)
{
    if (argc==1)              /* default: directory corrente */
        fsize(".");
    else
        while (--argc>0)
            fsize(*++argv);
    return 0;
}
```

La funzione `fsize` stampa la dimensione del file. Se il file è una directory, `fsize` chiama `dirwalk`, per gestire i file che tale directory contiene. Per decidere se il file è una directory, vengono usati i flag `S_IFMT` e `S_IFDIR`, definiti in `<sys/stat.h>`. La parentesizzazione è necessaria, perché la precedenza di `&` è infe-riore a quella di `==`.

```
int  stat(char *, struct stat *);
void  dirwalk(char *, void (*fcn)(char *));

/* fsize: stampa la dimensione del file "name" */
void  fsize(char *name)
{
    struct stat stbuf;

    if (stat(name, &stbuf)==-1)
    {
        fprintf(stderr, "fsize: non posso accedere a %s\n", name);
        return;
    }
    if ((stbuf.st_mode & S_IFMT)==S_IFDIR)
        dirwalk(name, fsize);
    printf("%8ld %s\n", stbuf.st_size, name);
}
```

La funzione `dirwalk` è una routine generale, che applica una funzione a ciascun file presente nella directo-ry specificata da `name`. Essa apre la directory, scandisce tutti i file in essa contenuti, chiamando su ognuno di essi la funzione desiderata, quindi chiude la directory e termina. Poiché `fsize` chiama `dirwalk` su ogni directory, le due funzioni si richiamano ricorsivamente.

```
#define  MAX_PATH  1024
```

```
/* dirwalk: applica fcn a tutti i file contenuti in dir */
void  dirwalk(char *dir, void (*fcn)(char *))
{
    char name[MAX_PATH];
    Dirent *dp;
    DIR *dfd;

    if ((dfd=opendir(dir))==NULL)
    {
        fprintf(stderr, "dirwalk: non posso aprire %s\n", dir);
        return;
    }
    while ((dp=readdir(dfd))!=NULL)
    {
        if (strcmp(dp->name, ".")==0  ||  strcmp(dp->name, "..")==0)
            continue;   /* tralascia se stessa e la directory padre */
        if (strlen(dir)+strlen(dp->name)+2>sizeof(name))
            fprintf(stderr, "dirwalk: il nome %s/%s
                    è troppo lungo", dir, dp->name);
        else
        {
            sprintf(name, "%s/%s", dir, dp->name);
            (*fcn)(name);
        }
    }
    closedir(dfd);
}
```

Ogni chiamata a `readdir` restituisce un puntatore alle informazioni relative al file successivo, oppure `NULL` se i file sono esauriti. Ogni directory contiene sempre una entry per se stessa, chiamata "`.`", ed una per la propria directory padre, "`..`"; queste due entry devono essere tralasciate, per evitare che il programma entri in un ciclo infinito.

Fino a questo livello, il codice è indipendente dal formato delle directory. Il passo successivo consiste nel presentare le versioni minimali di `opendir`, `readdir` e `closedir` per uno specifico tipo di file system. Le routine che seguono valgono per sistemi Version 7 e UNIX System V; esse utilizzano le informazioni sulle directory contenute in `<sys/dir.h>`, che sono le seguenti:

```
#ifndef  DIRSIZ
#define  DIRSIZ  14
#endif

struct direct                 /* entry della directory */
{
    ino_t d_ino;              /* numero dell'inode */
    char d_name[DIRSIZ];      /* un nome lungo non ha '\0' */
};
```

Alcune versioni consentono di avere nomi più lunghi e gestiscono strutture più complesse per le directory.

Il tipo `ino_t` è una `typedef` che descrive l'indice nella lista degli inode. Sui sistemi che usiamo normal-mente, questo tipo è un `unsigned short`, ma quest'informazione è bene non sia direttamente inclusa in un programma; questo tipo, infatti, potrebbe essere diverso su sistemi differenti, e ciò rende preferibile l'im-piego di `typedef`. L'insieme completo dei tipi "di sistema" è reperibile in `<sys/types.h>`.

`Opendir` apre la directory, verifica che il file sia una directory (questa volta usando la chiamata di sistema `fstat`, simile a `stat` ma che utilizza un descrittore di file), alloca una struttura per la directory e registra le informazioni:

```
int  fstat(int fd, struct stat *);

/* opendir: apre una directory per successive readdir */
DIR  *opendir(char *dirname)
{
    int fd;
```

```
    struct stat stbuf;
    DIR *dp;

    if ((fd=open(dirname, O_RDONLY, 0))==-1
         ||  fstat(fd, &stbuf)==-1
         ||  (stbuf.st_mode & S_IFMT)!=S_IFDIR
         ||  (dp=(DIR *)malloc(sizeof(DIR)))==NULL)
        return NULL;
    dp->fd=fd;
    return fd;
}
```

`closedir` chiude la directory e libera lo spazio:

```
/* closedir: chiude la directory aperta da opendir */
void closedir(DIR *dp)
{
    if (dp)
    {
        close(dp->fd);
        free(dp);
    }
}
```

Infine, `readdir` usa `read` per leggere ogni entry della directory. Se una di esse non è correntemente in uso (perché il file è stato rimosso), il suo numero di inode è zero, ed essa viene tralasciata. Altrimenti, il numero dell'inode ed il nome vengono collocati in una struttura `static`, ed all'utente viene fornito un puntatore a questa struttura. Ogni chiamata cancella le informazioni ricavate dalla chiamata precedente.

```
#include  <sys/dir.h>         /* struttura locale per la directory */

/* readdir: legge in sequenza le entry della directory */
Dirent  readdir(DIR *dp)
{
    struct direct dirbuf;     /* struttura locale per la directory */
    static Dirent d;          /* ritorno: struttura portabile */

    while (read(dp->fd, (char *) &dirbuf, sizeof(dirbuf))==sizeof(dirbuf))
    {
        if (dirbuf.d_ino==0)  /* entry non utilizzata */
            continue;
        d.ino=dirbuf.d_ino;
        strncpy(d.name, dirbuf.d_name, DIRSIZ);
        d.name[DIRSIZ]='\0';  /* assicura la terminazione */
        return &d;
    }
    return NULL;
}
```

Pur essendo alquanto specializzato, il programma `fsize` illustra un paio di idee importanti. Innanzitutto, molti programmi non sono "programmi di sistema"; essi si limitano ad usare informazioni mantenute dal si-stema operativo. Per questi programmi, è importante che la rappresentazione delle informazioni appaia sol-tanto negli header standard, inclusi dai programmi stessi al posto delle dichiarazioni esplicite. La seconda osservazione è che, facendo una certa attenzione, è possibile creare un'interfaccia abbastanza indipenden-te dal sistema verso oggetti che invece dipendono da esso. Le funzioni della libreria standard costituiscono, a questo proposito, degli ottimi esempi.

Esercizio 8.5 Modificate il programma `fsize` in modo che stampi anche le altre informazioni contenute nell'inode.

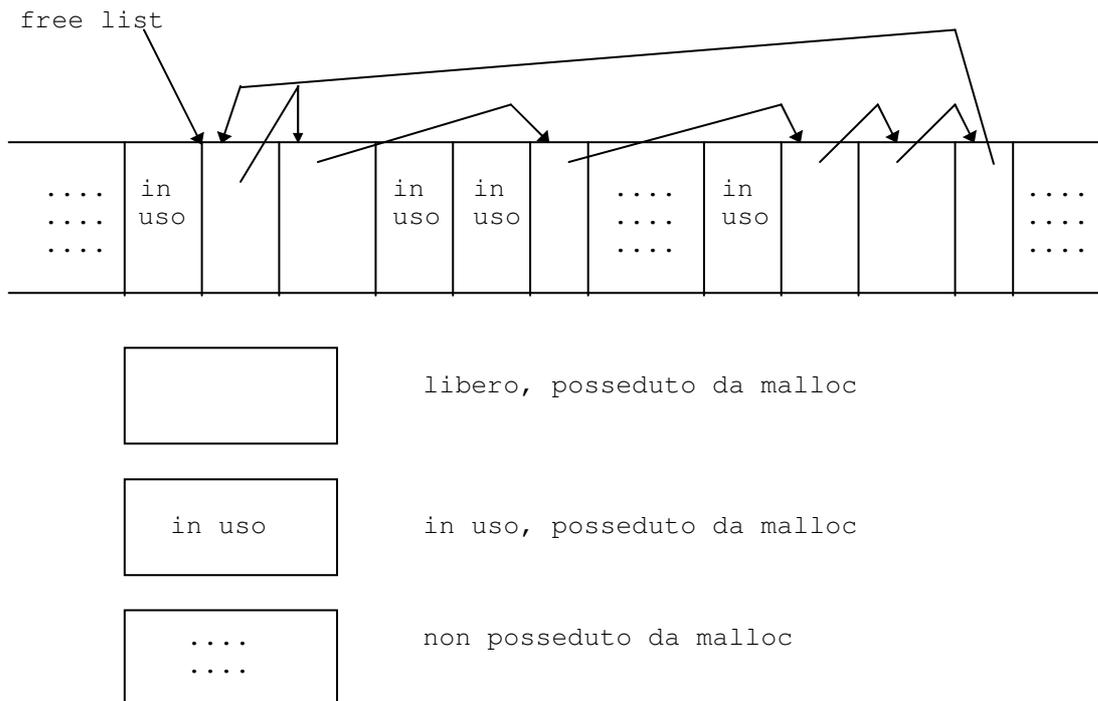
8.7 Esempio – Un Allocatore di Memoria

Nel Capitolo 5 abbiamo presentato un allocatore di memoria molto limitato. La versione che scriveremo ora è invece priva di limitazioni. Le chiamate a `malloc` e `free` possono essere effettuate in un ordine qualsia-si;

per ottenere la memoria necessaria, `malloc` invoca il sistema operativo. Queste routine illustrano alcu-ne delle considerazioni legate alla stesura di codice machine-dependent secondo criteri che lo rendono ab-bastanza indipendente dal sistema usato; inoltre, esse mostrano un caso reale di utilizzo delle strutture, del-le `union` e delle `typedef`.

Invece di allocarlo a partire da un vettore di ampiezza prefissata, `malloc` preleva lo spazio di memoria dal sistema operativo. Poiché, nel programma, possono esistere altre attività che richiedono aree di memoria senza invocare questo allocatore, lo spazio gestito da `malloc` può non essere contiguo. Per questo motivo, la memoria libera viene gestita sotto forma di una lista di blocchi liberi (free list). Ogni blocco contiene una ampiezza, un puntatore al blocco successivo e l'area vera e propria. I blocchi sono ordinati in modo cre-scente rispetto al loro indirizzo, e l'ultimo blocco (cioè quello con indirizzo più alto) punta al primo.

All'atto di una richiesta di memoria, la free list viene scandita fino al reperimento di un blocco sufficiente-mente grande. Questo algoritmo è detto "first fit", in contrapposizione all'algoritmo "best fit", che cerca il più piccolo blocco in grado di soddisfare la richiesta. Se la dimensione del blocco trovato coincide con quella ri-chiesta, il blocco viene staccato dalla lista e fornito all'utente. Se il blocco è troppo grande, esso viene spez-zato, ed all'utente viene restituita soltanto la parte che egli ha richiesto, mentre l'altra parte rimane nella li-sta libera. Se non esiste alcun blocco sufficientemente grande da poter soddisfare la richiesta, la memoria necessaria viene allocata dal sistema operativo ed inserita nella lista libera.

Anche il rilascio di un blocco provoca una scansione della lista, allo scopo di trovare la posizione corretta nella quale inserire il blocco liberato. Se quest'ultimo è adiacente ad un blocco già libero, i due blocchi ven-gono unificati, in modo da ridurre la frammentazione della memoria. L'identificazione di due blocchi adia-centi è facilitata dal fatto che la free list è ordinata in ordine crescente.

Un problema, al quale abbiamo accennato nel Capitolo 5, consiste nell'assicurare che la memoria ritornata da `malloc` sia allineata in modo corretto rispetto al tipo di oggetti che deve contenere. Nonostante le mac-chine possano variare, ogni macchina possiede un tipo più restrittivo: se esso può essere memorizzato ad un particolare indirizzo, anche tutti gli altri possono esserlo. Su alcune macchine, il tipo più restrittivo è il `double`, mentre su altre è l'`int` o il `long`.

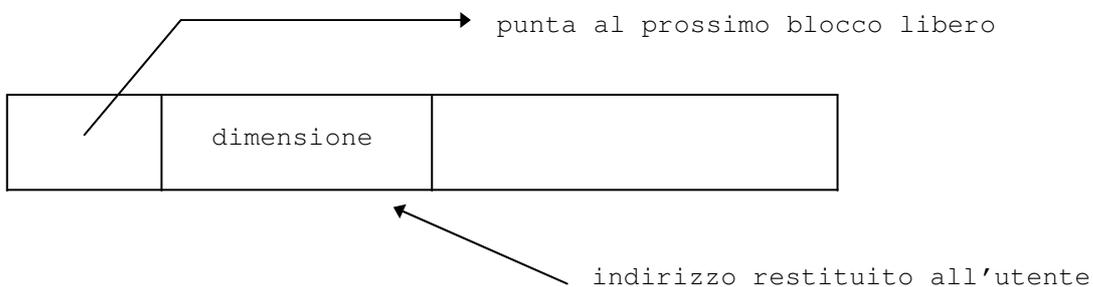
Un blocco libero contiene un puntatore al blocco successivo della catena, l'ampiezza del blocco e lo spazio stesso; l'informazione di controllo collocata all'inizio è detta "header". Per semplificare l'allineamento, tutti i blocchi sono multipli della dimensione dell'header, e l'header è allineato in modo opportuno. Tutto ciò può essere fatto con una `union`, contenente la struttura dell'header ed un'istanza del tipo più restrittivo, che as-sumiamo essere il `long`:

```
typedef long Align;          /* per allineare alla long */
```

```
union header {                  /* header del blocco */
    struct {
        union header *ptr;      /* blocco successivo in free list */
        unsigned size;          /* dimensione di questo blocco */
    }s;
    Align x;                    /* forza l'allineamento dei blocchi */
};

typedef union header Header;
```

Il campo `Align` non viene mai usato; il suo scopo è soltanto quello di allineare ogni header.

Nella funzione `malloc` l'ampiezza richiesta, in caratteri, viene arrotondata al corretto multiplo dell'ampiezza dell'header; il blocco che verrà allocato contiene un elemento in più, destinato a contenere l'header stesso, e questo valore è quello contenuto nel campo `size` dell'header. Il puntatore fornito da `malloc` punta allo spazio libero, non all'header. L'utente può utilizzare in qualsiasi modo l'area ottenuta, ma scrivere al di fuori di essa equivale a distruggere l'ordine della lista.

Un blocco restituito da `malloc`

Il campo contenente l'ampiezza è necessario, perché lo spazio gestito da `malloc` può non essere contiguo, e quindi non è possibile calcolare le ampiezze sfruttando l'aritmetica dei puntatori.

La variabile `base` viene usata per innescare il meccanismo. Se `freep` è `NULL`, come nel caso della prima chiamata a `malloc`, allora viene creata una free list degenere, contenente un blocco di ampiezza zero che punta a se stesso. In ogni caso, la free list viene sempre scandita. La ricerca di un blocco libero di ampiezza adeguata inizia dal punto (`freep`) in cui era stato trovato l'ultimo blocco; questa strategia aiuta a mantenere omogenea la lista. Se viene trovato un blocco troppo grande, all'utente viene restituita la sua parte finale; in tal modo è necessario aggiornare soltanto il campo `size` dell'header originario. In ogni caso, il puntatore fornito all'utente punta allo spazio libero del blocco, che inizia un'unità dopo l'header.

```
static Header base;             /* lista vuota, per iniziare */
static Header *freep=NULL;      /* inizio della free list */

/* malloc: allocatore di memoria di uso generale */
void  *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *morecore(unsigned);
    Unsigned nunits;

    nunits=(nbytes+sizeof(Header)-1)/sizeof(Header)+1;
    if ((prevp=freep)==NULL)        /* non esiste free list */
    {
        base.s.ptr=freep=prevp=&base;
        base.s.size=0;
    }
    for (p=prevp->s.ptr; ; prevp=p, p=p->s.ptr)
    {
        if (p->s.size>=nunits)      /* spazio sufficiente */
        {
            if (p->s.size==nunits)  /* esattamente */
```

```
                prevp->s.ptr=p->s.ptr;
            else                    /* alloca la parte finale */
            {
                p->s.size-=nunits;
                p+=p->s.size;
                p->s.size=nunits;
            }
            freep=prevp;
            return (void *)(p+1);
        }
    if (p==freep)                   /* la free list è terminata */
        if ((p=morecore(nunits))==NULL)
            return NULL;            /* non c'è più spazio */
}
```

La funzione `morecore` ottiene memoria dal sistema operativo. I dettagli di come ciò avviene variano da si-stema a sistema. Poiché la richiesta di memoria al sistema operativo è un'operazione relativamente costo-sa, noi non vogliamo effettuarla ad ogni chiamata di `malloc`, quindi facciamo in modo che `morecore` allo-chi almeno `NALLOC` unità; questo blocco verrà successivamente separato in blocchi più piccoli, in base alla necessità del momento. Dopo avere inizializzato il campo dell'ampiezza, `morecore` inserisce in free list la memoria aggiuntiva, invocando `free`.

In UNIX, la chiamata di sistema `sbrk(n)` ritorna un puntatore a `n` byte aggiuntivi di memoria. `sbrk` ritorna `-1` se lo spazio richiesto non è disponibile. Per poterlo confrontare con il valore di ritorno, `-1` deve essere forzato a `char *`. Ancora una volta, il casting rende la funzione abbastanza indipendente dalla rappresenta-zione adottata per i puntatori sulle diverse macchine. Notiamo che la nostra `malloc` assume che il confron-to fra due puntatori a blocchi diversi ottenuti con `sbrk` sia significativo. Questo non è garantito dallo stan-dard, che consente soltanto il confronto fra puntatori all'interno di uno stesso vettore. Per questo motivo, questa versione di `malloc` è portabile soltanto su macchine che consentono il confronto generalizzato fra puntatori.

```
#define  NALLOC  1024         /* numero minimo di unità richieste */

/* morecore: chiede al sistema memoria aggiuntiva */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu<NALLOC)
        nu=NALLOC;
    cp=sbrk(nu*sizeof(Header));
    if (cp==(char *) -1)      /* non c'è spazio */
        return NULL;
    up=(Header *) cp;
    up->s.size=nu;
    free((void *)(up+1));
    return freep;
}
```

L'ultima funzione che dobbiamo scrivere è `free`. Essa scandisce la free list, partendo da `freep`, e cerca la posizione nella quale inserire il blocco. Tale posizione può trovarsi in due blocchi esistenti oppure ad uno degli estremi della lista. In ogni caso, se il blocco che dev'essere rilasciato è adiacente ad uno già esistente, i due blocchi contigui vengono uniti. L'unico problema consiste nel mantenere correttamente aggiornati i puntatori e le dimensioni.

```
/* free: inserisce in free list il blocco ap */
void  free(void *ap)
{
    Header *bp, *p;

    bp=(Header *)ap-1;        /* punta all'header del blocco */
    for (p=freep; !(bp>p  &&  bp<p->s.ptr); p=p->s.ptr)
        if (p>=p->s.ptr  &&  (bp>p  ||  bp<p->s.ptr))
            break;    /* il blocco liberato è ad un estremo della lista */
```

```
        if (bp+bp->s.size==p->s.ptr)
        {                           /* lo unisce al blocco successivo */
            bp->s.size+=p->s.ptr->s.size;
            bp->s.ptr=p->s.ptr->s.ptr;
        }
        else
            bp->s.ptr=p->s.ptr;
        if (p+p->s.size==bp)
        {                           /* lo unisce al blocco precedente */
            p->s.size+=bp->s.size;
            p->s.ptr=bp->s.ptr;
        }
        else
            p->s.ptr=bp;
        freep=p;
    }
```

Anche se l'allocazione di memoria è intrinsecamente machine-dependent, il codice delle precedenti funzioni illustra come le dipendenze dalla macchina possano essere controllate e confinate a segmenti molto limitati di un programma. L'uso di `typedef` e di `union` consente di gestire l'allineamento (purché `sbrk` fornisca un puntatore appropriato). L'operatore di casting consente di esplicitare le conversioni dei puntatori. Anche se i dettagli di questo esempio sono strettamente legati all'allocazione di memoria, l'approccio generale rimane valido anche per molte altre situazioni tipiche.

Esercizio 8.6 La funzione `calloc(n, size)`, della libreria standard, ritorna un puntatore ad `n` oggetti di ampiezza pari a `size`, ed inizializza a zero la memoria allocata. Scrivete `calloc`, invocando `malloc` op-pure modificandola.

Esercizio 8.7 `malloc` accetta in input una dimensione, senza effettuare su di essa alcun controllo; `free` assume che il blocco da liberare contenga un campo `size` significativo. Migliorate queste routine introdu-cendo il controllo delle condizioni di errore.

Esercizio 8.8 Scrivete una routine `bree(p, n)` che libera un arbitrario blocco `p` di `n` caratteri e lo inserisce in una free list gestita da `malloc` e `free`. Usando `bfree`, un utente può aggiungere alla free list, in qual-siasi istante, un vettore statico o esterno.

APPENDICE A

REFERENCE MANUAL

A1. Introduzione

Questo manuale descrive il linguaggio C come specificato dal “Draft Proposed American National Standard for Information System – Programming Language C”, documento numero X3J11/88-001, datato 11 Gennaio 1988. Questo documento non contiene lo standard definitivo, ed è ancora possibile che vengano apportate modifiche al linguaggio. Di conseguenza, questo manuale non può essere considerato una descrizione definitiva del linguaggio. Inoltre, esso è in realtà un’interpretazione dello schema standard proposto, e non lo standard stesso.

Nella maggior parte dei casi, questo manuale segue la linea del Draft Standard, che a sua volta rispecchia quella della prima edizione di questo libro, anche se i dettagli organizzativi sono diversi. Se si escludono la ridenominazione di alcune produzioni e la mancata formalizzazione della definizione dei token lessicali o del preprocessor, la grammatica presentata in questo manuale può essere considerata equivalente a quella del draft.

> In questo manuale, i commenti sono indentati e scritti in caratteri più piccoli, come questi. Spesso questi commenti evidenziano i punti nei quali l’ANSI C differisce dal linguaggio definito nella prima edizione di questo libro, o da raffinamenti prodotti in un secondo tempo nei diversi compilatori.

A2. Convenzioni Lessicali

Un programma è costituito da una o più *unità di traduzione* registrate nei file. Esso viene tradotto attraverso diverse fasi, descritte nel paragrafo A12. Le prime fasi effettuano trasformazioni lessicali di basso livello, eseguendo le direttive introdotte da linee che iniziano con il carattere `#`, ed effettuando la definizione e la espansione delle macro. Quando il preprocessing descritto nel paragrafo A12 termina, il programma è ridot-to ad una sequenza di token.

A2.1 Token

Esistono sei classi di token : identificatori, parole chiave, costanti, stringhe letterali, operatori ed altri sepa-ratori. Gli spazi, i caratteri di tabulazione verticali ed orizzontali, i new line, i salti pagina ed i commenti (glo-balmente definibili come “spazi bianchi”), nel seguito vengono descritti e considerati soltanto in qualità di separatori di token. Gli spazi sono infatti necessari per separare identificatori, parole chiave e costanti che altrimenti sarebbero adiacenti.

Se il flusso di input è stato separato in token fino ad un certo carattere, il token successivo è la più lunga stringa di caratteri che può costituire un token.

A2.2 Commenti

I caratteri `/*` introducono un commento, che termina con i caratteri `*/`. I commenti non possono essere nidificati e non possono comparire all’interno di stringhe.

A2.3 Identificatori

Un identificatore è una sequenza di lettere e cifre. Il primo carattere dev’essere una lettera, fra le quali è compreso il carattere underscore _. Le lettere maiuscole sono considerate diverse da quelle minuscole. Gli identificatori possono avere qualsiasi lunghezza e, per gli identificatori interni, sono significativi almeno i pri-mi 31 caratteri ; questo limite, in alcune implementazioni, è superiore. Gli identificatori interni comprendono i nomi delle macro di preprocessor e tutti gli altri nomi privi di linkaggio esterno (paragrafo A11.2). Gli identificatori con linkaggio esterno sono soggetti a restrizioni maggiori: le diverse implementazioni possono considerare significativi anche solo i primi 6 caratteri, e possono ignorare la distinzione fra lettere maiuscole e minuscole.

A2.4 Parole Chiave

Quelli che seguono sono identificatori riservati per essere utilizzati come parole chiave, e non possono es-sere usati altrimenti:

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

Alcune implementazioni riservano anche le parole chiave `fortran` e `asm`.

Le parole chiave `const`, `signed` e `volatile` sono state introdotte con l'ANSI C; `enum` e `void` sono nuove rispetto alla prima edizione, anche se comunemente usate da tempo; `entry`, che finora era riservata ma non veniva mai utilizzata, è stata eliminata.

A2.5 Costanti

Esistono diverse classi di costanti. Ognuna di esse ha un proprio tipo di dati; il paragrafo A4.2 illustri i tipi principali.

costante:
    costante-intera
    costante-carattere
    costante-floating
    costante-enumerativa

A2.5.1 Costanti Intere

Una costante intera, costituita da una sequenza di cifre, viene considerata ottale se la prima cifra è `0`, deci-male altrimenti. Le costanti ottali non contengono le cifre `8` e `9`. Una sequenza di cifre preceduta da `0x` e `0X` viene considerata una costante intera esadecimale. Le cifre esadecimali comprendono le lettere da `a` o `A` alle lettere `f` o `F`, estremi inclusi, che rappresentano i valori da 10 a 15.

Una costante intera può contenere il suffisso `u` oppure `U`, che specifica che essa è priva di segno. Un suffis-so `l` o `L` indica invece che la costante è un `long`.

Il tipo di una costante intera dipende dal suo formato, dal suo valore e dal suffisso (si veda il paragrafo A4 per una discussione sui tipi).Se la costante è decimale e priva di suffissi, essa ha il primo, fra i tipi seguenti, con cui può essere rappresentato il suo valore: `int`, `long int`, `unsigned long int`. Se la costante è priva di suffissi ed è ottale o esadecimale, il suo tipo è il primo possibile tra: `int`, `unsigned int`, `long int`, `unsigned long int`. Se la costante ha il suffisso `u` (oppure `U`), il suo tipo è `unsigned int` o `unsigned long int`. Se è presente il suffisso `l` (o `L`), il tipo è `long int` oppure `unsigned long int`.

L'elaborazione del tipo delle costanti intere è notevolmente più sofisticata rispetto alla prima edizione, che si limitava a considerare `long` le costanti con valore intero molto grande. I suffissi `u` e `U` sono nuovi.

A2.5.2 Costanti Carattere

Una costante carattere è una sequenza di uno o più caratteri racchiusi fra singoli apici, come `'x'`. Il valore di una costante composta da un solo carattere è il valore numerico del carattere all'interno del set di caratteri della macchina. Il valore di una costante composta da più caratteri è definito dall'implementazione.

Le costanti carattere non contengono il carattere `'` (apice singolo) ed i new line; per rappresentare questi caratteri, ed alcuni altri, si possono utilizzare le seguenti sequenze di escape:

```
newline              NL(LF)      \n
```

```
tab orizzontale        HT        \t
tab verticale          VT        \v
backspace              BS        \b
return                 CR        \r
salto pagina           FF        \f
allarme udibile        BEL       \a
backslash              \         \\
punto interrogativo    ?         \?
apice singolo          ’         \’
apice doppio           “         \”
numero ottale          ooo       \ooo
numero esadecimale     hh        \hh
```

La sequenza `\`*ooo* è composta da un carattere backslash seguito da 1, 2 o 3 cifre ottali, che specificano il valore del carattere desiderato. Un esempio comune di costrutto di questo tipo è la sequenza `\`*0* (una sola cifra) che indica il carattere *NUL.* La sequenza `\`*xhh* è costituita da un carattere backslash, da una `x` e da ci-fre esadecimali, che specificano il valore del carattere desiderato. Non c’è limite al numero di cifre, ma il comportamento è indefinito se il valore risultante supera quello massimo contemplato dal set di caratteri della macchina. Per le sequenze di escape ottali ed esadecimali, se la macchina considera il tipo `char` comprensivo di segno, il valore viene esteso con il segno, come se venisse applicato l’operatore di cast.

In alcune implementazioni esiste un set di caratteri esteso, che non può essere rappresentato con il tipo `char`. Una costante appartenente ad un set di questo tipo viene preceduta dal carattere `L`, per esempio `L’x’`, e viene chiamata costante carattere estesa. Essa è di tipo `wchar_t`, un tipo intero definito nell’hea-der standard `<stddef.h>`. Anche per queste costanti si possono utilizzare le sequenze di escape ottali o esadecimali; l’effetto è indefinito se il valore specificato eccede quello massimo rappresentabile con il tipo `wchar_t`.

Alcune di queste sequenze di escape, come per esempio la rappresentazione esadecimale di un carattere sono nuove. Anche le costanti carattere estese sono una novità. Il set di caratteri normalmente usato in America e nell’Europa occidentale può essere rappresentato con il tipo `char`; lo scopo principale dell’introduzione del tipo `wchar_t` è la rappresentazione dei linguaggi asiatici.

A2.5.3 Costanti Floating

Una costante floating consiste in una parte intera, un punto decimale, una parte frazionaria, una `e` oppure una `E`, un esponente intero con un segno (opzionale) ed un suffisso opzionale per il tipo; `f`, `F`, `l` o `L`. Le par-ti intera e frazionaria sono sequenze di cifre. Una delle due (non entrambe) può essere omessa; possono essere tralasciati (non contemporaneamente) l’esponente `e` oppure il punto decimale. Il tipo è determinato dal suffisso; `F` o `f` indica un `float`, `l` o `L` indica un `long double`; se il suffisso non è presente, il tipo è `double`.

I suffissi sulle costanti floating sono nuovi.

A2.5.4 Costanti Enumerative

Gli identificatori dichiarati come enumeratori (paragrafo A8.4) sono costanti di tipo `int`.

A2.6 Stringhe Letterali

Una stringa letterale, detta anche stringa costante, è una sequenza di caratteri racchiusi fra doppi apici, come `”...”`. Una stringa è un “vettore di caratteri”, appartiene alla classe di memoria `static` (paragrafo A4) e viene inizializzata con i caratteri dati. Il fatto che due stringhe identiche siano considerate distinte di-pende dall’implementazione, ed il comportamento di un programma che tenta di modificare una stringa letterale è indefinito.

Stringhe letterali adiacenti vengono concatenate in una singola stringa. Dopo ogni concatenazione, viene inserito un byte nullo (`\0`) che consente ai programmi di scorrere la stringa ed individuarne la fine. Le strin-ghe letterali non contengono new line e doppi apici; questi caratteri sono rappresentabili con le stesse se-quenze di escape descritte per le costanti carattere.

Come per le costanti carattere, anche le stringhe letterali, in un set di caratteri esteso, possono essere pre-cedute da una L, come L"...". Le stringhe letterali estese sono di tipo "vettore di wchar_t". Il risultato della concatenazione fra stringhe letterali estese e stringhe normali è indefinito.

> Il fatto che due stringhe letterali possano non essere distinte ed il divieto di modificarle sono stati introdotti con l'ANSI C, così come la concatenazione di stringhe letterali adiacenti. Anche le stringhe letterali estese sono una novità.

A3. Notazioni Sintattiche

Nella notazione adottata da questo manuale, le categorie sintattiche sono indicate dallo stile *corsivo*, mentre per le parole ed i caratteri viene usato lo stile typewriter. Le categorie alternative sono nor-malmente disposte su linee diverse; in alcuni casi, un vasto insieme di alternative obbligate viene presenta-to su una unica riga, iniziata dalla frase "uno fra". Un simbolo opzionale, terminale o meno, ha il suffisso "*opt*", in modo che, per esempio,

{espressione$_{opt}$ }

indica un'espressione opzionale, racchiusa fra parentesi graffe. La sintassi è riassunta nel paragrafo A13.

> Diversamente da quanto accadeva nella grammatica data nella prima edizione di questo libro, quella presentata qui rende esplicita la precedenza e l'associatività degli operatori nelle espressioni.

A4. Significato degli Identificatori

Gli identificatori, o nomi, si riferiscono a diverse classi di oggetti: funzioni; tag delle strutture, delle union e delle enumerazioni; membri di strutture o union; costanti enumerative; nomi di tipi definiti con typedef e oggetti. Un oggetto, talvolta chiamato variabile, è una locazione di memoria, e la sua interpretazione dipen-de da due attributi principali: la sua *classe di memoria* ed il suo *tipo*. La classe di memoria determina la du-rata della vita della memoria associata all'oggetto identificato; il tipo determina il significato dei valori trovati nell'oggetto identificato. Un nome ha anche uno scope, che è la regione del programma nella quale tale no-me è conosciuto, ed un linkaggio, che determina se lo stesso nome, in un altro scope, si riferisce allo stesso oggetto o funzione. Lo scope ed il linkaggio sono discussi nel paragrafo A11.

A4.1 Classi di Memoria

Esistono due classi di memoria: automatica e statica. Alcune parole chiave, insieme al contesto della dichiarazione di un oggetto, ne specificano la classe di appartenenza. Gli oggetti automatici sono locali ad un blocco (paragrafo A9.3), all'uscita del quale vengono abbandonati. Se non comprendono alcuna specifi-cazione della classe, oppure se compare lo specificatore auto, le dichiarazioni all'interno di un blocco creano oggetti automatici. Gli oggetti dichiarati register sono automatici e, se possibile, vengono allocati nei registri della macchina, che hanno un'elevata velocità di accesso.

Gli oggetti statici possono essere locali ad un blocco od esterni a qualsiasi blocco, ma in entrambi i casi mantengono il loro valore anche fra l'uscita ed il successivo rientro da funzioni e blocchi. All'interno di un blocco, compreso quello che costituisce il corpo di una funzione, gli oggetti statici sono dichiarati esterna-mente a qualsiasi blocco, cioè allo stesso livello delle definizioni di funzione, sono sempre statici. Essi pos-sono essere resi locali ad una particolare unità di traduzione tramite la parola chiave static; una simile operazione attribuisce agli oggetti un *linkaggio interno*. Omettendo lo specificatore della classe di memoria, oppure utilizzando la parola chiave extern, questi oggetti diventano globali all'intero programma, ed acqui-stano così un *linkaggio esterno*.

A4.2 Tipi Fondamentali

Esistono diversi tipi fondamentali. L'header standard <limits.h>, descritto nell'Appendice B, definisce i valori massimi e minimi di ogni tipo, in relazione all'implementazione specifica. I numeri dati nell'Appendice B mostrano le grandezze minime consentite.

Gli oggetti dichiarati come caratteri (char) sono sufficientemente grandi da potere rappresentare qualsiasi membro del set locale di caratteri. Se un carattere di questo set viene memorizzato in un oggetto di tipo char, il suo valore equivale al codice intero di quel carattere, ed è non negativo. Nelle variabili di tipo char

possono essere registrate anche altre qualità, ma la sfera dei valori disponibili, e soprattutto il fatto che il valore abbia o meno il segno, dipendono dall'implementazione.

I caratteri privi di segno, dichiarati `unsigned char`, occupano la stessa quantità di spazio occupata dai ca-ratteri normali ma, a differenza di questi, sono sempre non negativi; anche i caratteri dichiarati `signed char` occupano uno spazio uguale a quello occupato dai caratteri normali.

> Il tipo `unsigned char` non compariva nella prima edizione di questo libro, pur essendo comunemente utilizzato. Il tipo `signed char`, invece, è nuovo.

Oltre al tipo `char`, sono disponibili fino a tre tipi di interi: `short int`, `int` e `long int`. Gli oggetti di tipo `int` hanno la dimensione naturale suggerita dall'architettura della macchina; le altre ampiezze vengono fornite per soddisfare esigenze particolari. Gli interi più ampi riservano un'area di memoria almeno pari a quella degli interi più piccoli, ma l'implementazione può rendere equivalente il tipo `int` al tipo `short int` o al tipo `long int`. A meno di direttive esplicite particolari, tutti i tipi `int` rappresentano oggetti con segno.

Gli interi privi di segno, dichiarati usando la parola chiave `unsigned`, seguono le leggi dell'aritmetica modu-lo 2^*n*, dove *n* è il numero di bit della rappresentazione; quindi le operazioni aritmetiche su oggetti privi di segno non possono mai andare in overflow. L'insieme dei valori non negativi rappresentabili con un oggetto con segno è un sottoinsieme dei valori che possono essere registrati nel corrispondente oggetto privo di se-gno, e la rappresentazione dei valori rappresentabili in entrambe le classi è la stessa.

Anche i tipi floating point in singola (`float`), in doppia (`double`) o in extra (`long double`) precisione pos-sono essere sinonimi, ma ogni elemento di questa lista dev'essere preciso almeno quanto quelli che lo precedono.

> Il tipo `long double` è nuovo. La prima edizione considerava equivalenti i tipi `double` e `long float`; quest'ultima locuzione è stata eliminata.

Le *enumerazioni* sono di un unico tipo ed hanno valori interi; ad ogni enumerazione è associato un insieme di nomi costanti (paragrafo A8.4). Le enumerazioni si comportano come gli interi, anche se è abbastanza comune che i compilatori generino un messaggio di warning quando ad un oggetto di particolare tipo enu-merativo viene assegnato qualcosa di diverso dalle sue costanti o da un'espressione del suo stesso tipo.

Poiché gli oggetti di questi tipi possono essere interpretati come numeri, noi ci riferiremo ad essi come ai tipi *aritmetici*. I tipi `char` e `int` (di ogni dimensione), con o senza segno, uniti ai tipi enumerativi verranno chiamati, nel loro complesso, tipi *interi*. I tipi `float`, `double` e `long double` verranno chiamati tipi *floating*.

Il tipo `void` specifica un insieme vuoto di valori. Esso viene utilizzato come tipo restituito da funzioni che non generano alcun valore.

A4.3 Tipi Derivati

Oltre ai tipi fondamentali, esiste una classe concettualmente infinita di tipi derivati, costruiti a partire dai tipi fondamentali nei seguenti modi:

vettori di oggetti di un certo tipo;
funzioni che ritornano oggetti di un certo tipo;
puntatori ad oggetti di un certo tipo;
strutture composte da una sequenza di oggetti di tipo diverso;
union, in grado di contenere uno fra diversi oggetti di diverso tipo.

In generale, questi metodi di costruzione degli oggetti possono essere utilizzati ricorsivamente.

A4.4 Qualificatori di Tipo

Il tipo di un oggetto può avere alcuni qualificatori aggiuntivi. Dichiarare un oggetto come `const` significa che il suo valore non verrà modificato; dichiararlo `volatile` significa, invece, che esso possiede proprietà particolari, significative per l'ottimizzazione. Nessuno di questi qualificatori altera la sfera dei valori o le pro-prietà aritmetiche dell'oggetto. I qualificatori vengono discussi nel paragrafo A8.2.

A5. Oggetti e Lvalue

Un *oggetto* è un'area di memoria alla quale è stato associato un nome; un *lvalue* è un'espressione che si riferisce ad un oggetto. Un esempio ovvio di espressione lvalue è un identificatore avente un tipo ed una classe di memoria significativi. Esistono operatori che producono degli lvalue: per esempio, se `E` è una espressione di tipo puntatore, allora `*E` è un'espressione lvalue, che si riferisce all'oggetto puntato da `E`. Il nome "lvalue" deriva dall'espressione di assegnamento `E1=E2`, nella quale l'operando sinistro ("left") deve essere un'espressione lvalue. La discussione di ogni operatore specifica se esso si aspetta operandi lvalue e se produce un lvalue.

A6. Conversioni

In base ai loro operandi, alcuni operatori possono provocare la conversione di un operando da un tipo all'al-tro. Questa sezione illustra il risultato che ci si deve attendere da queste conversioni. Il paragrafo A6.5 rias-sume le conversioni eseguite dagli operatori più usati; in caso di necessità, esso potrà essere integrato con la lettura della discussione relativa al singolo operatore.

A6.1 Trasformazione di Interi

Un carattere, un piccolo intero o un field intero, tutti con o senza segno, oppure un oggetto di tipo enumerativo, possono essere utilizzati, in un'espressione, in tutte le posizioni nelle quali può ricorrere un intero. Se un `int` può rappresentare tutti i valori del tipo originario, allora il valore viene convertito in un `int`; in caso contrario, il valore viene convertito in un `unsigned int`. Questo processo è detto *trasforma-zione degli interi*.

A6.2 Conversione di Interi

Qualsiasi intero viene convertito in un dato tipo privo di segno trovando il più piccolo valore non negativo congruente a quell'intero, maggiore di uno, in modulo, rispetto al massimo valore rappresentabile nel tipo privo di segno. In una rappresentazione in complemento a due, questo significa troncare a sinistra l'intero, se il tipo privo di segno è più piccolo, ed inserire degli zeri per i valori privi di segno e applicare l'estensione del segno a quelli con segno se il tipo `unsigned` è più ampio. Quando un intero viene convertito in un tipo con segno, il suo valore rimane inalterato se può essere rappresentato nel nuovo tipo; in caso contrario, esso è definito dall'implementazione.

A6.3 Interi e Floating

Quando un valore di tipo floating viene convertito in un tipo intero, la parte frazionaria viene scartata; se il valore risultante non può essere rappresentato nel tipo intero voluto, il comportamento è indefinito. In particolare, non è specificato il risultato della conversione di valori floating negativi in tipi interi privi di segno.

Quando un valore di tipo intero viene convertito in un floating, ed il valore è rappresentabile ma non esatta-mente, il risultato può essere sia il primo valore rappresentabile inferiore a quello dato, sia il primo superio-re. Se il risultato non appartiene alla sfera dei valori rappresentabili, il comportamento è indefinito.

A6.4 Tipi Floating

Quando un valore floating viene convertito in un altro floating, di precisione pari o superiore a quella di par-tenza, il valore resta invariato. Quando la precisione del tipo finale è inferiore a quella di partenza, ed il valore è rappresentabile, il risultato può essere sia il primo valore rappresentabile inferiore che quello supe-riore. Se il risultato non è rappresentabile, il comportamento è indefinito.

A6.5 Conversione Aritmetiche

Molti operatori provocano delle conversioni. L'effetto è quello di ridurre gli operandi ad un tipo comune, che è anche il tipo del risultato. Questo procedimento è detto *conversione aritmetica usuale*.

In primo luogo, se uno degli operandi è un `long double`, l'altro viene convertito in un `long dou-ble`.
Altrimenti, se uno degli operandi è un `double`, anche l'altro viene convertito in un `double`.
Altrimenti, se uno degli operandi è un `float`, anche l'altro viene convertito in un `float`.
Altrimenti, su entrambi gli operandi viene eseguita la trasformazione degli interi; quindi, se uno di essi è un `unsigned long int`, anche l'altro lo diventa.
Altrimenti, se uno degli operandi è un `long int` e l'altro è un `unsigned int`, l'effetto dipende dal fatto che un `long int` possa rappresentare tutti i valori di un `unsigned int`; se questo avviene, lo operando `unsigned int` viene convertito in `long int`; in caso contrario, entrambi vengono trasfor-mati in `unsigned long int`.
Altrimenti, se un operando è un `long int`, anche l'altro viene convertito in un `long int`.
Altrimenti, se un operando è un `unsigned int`, anche l'altro viene convertito in un `unsigned int`.
Altrimenti, entrambi gli operandi sono di tipo `int`.

A questo proposito sono state fatte due modifiche. In primo luogo, l'aritmetica sugli operandi `float` può avvenire in singola pre-cisione, invece che in doppia; la prima edizione specificava che tutta l'aritmetica sui floating lavorava in doppia precisione. In secondo luogo, i tipi privi di segno più piccoli, quando vengono combinati con interi più grandi, non propagano il loro essere privi di segno al tipo risultante; nella prima edizione, il tipo privo di segno dominava sempre. Le nuove regole sono leggermente più complesse, ma riducono in un certo senso le "sorprese" che si possono avere quando una quantità priva di segno opera insieme ad una con segno. Risultati inattesi si possono ancora verificare, quando un'espressione priva di segno viene confrontata con una con segno della stessa dimensione.

A6.6 Puntatori ed Interi

Un'espressione di tipo intero può essere sommata o sottratta da un puntatore; in questo caso, l'espressione intera viene convertita in base alle regole descritte per l'operatore di addizione (paragrafo A7.7).

Due puntatori ad oggetti di medesimo tipo, appartenenti allo stesso vettore, possono essere sottratti; il risul-tato viene convertito in un intero, secondo quando specificato per l'operatore di sottrazione (paragrafo A7.7). Un'espressione costante intera con valore 0, o un'espressione di questo tipo forzata al tipo `void *`, può essere convertita per mezzo di un casting, di un assegnamento o di un confronto, ad un puntatore di qualsiasi tipo. Questo produce un puntatore nullo uguale ad un puntatore nullo dello stesso tipo, ma diverso da qualsiasi puntatore ad una funzione o ad un oggetto.

Sui puntatori sono consentite alcune altre conversioni, che comprendono però degli aspetti legati all'imple-mentazione. Queste conversioni devono essere specificate tramite un operatore esplicito di conversione di tipo, oppure tramite un casting (paragrafi A7.5 e A8.8).

Un puntatore può essere convertito in un tipo intero sufficientemente grande da contenerlo; l'ampiezza ne-cessaria e la funzione di trasformazione dipendono dall'implementazione.

Un oggetto di tipo intero può essere esplicitamente convertito in un puntatore. La trasformazione garantisce che un intero ricavato da un puntatore venga sempre ricondotto al puntatore di partenza, ma è dipendente dall'implementazione negli altri casi.

Un puntatore ad un tipo può essere convertito ad un puntatore ad un altro tipo. Il puntatore risultante può provocare errori di indirizzamento se il puntatore di partenza non si riferisce ad un oggetto correttamente al-lineato in memoria. È garantito che un puntatore ad un tipo possa essere fatto puntare ad un tipo soggetto a restrizioni di allineamento almeno pari a quelle del tipo dell'oggetto puntato inizialmente; la nozione di "alli-neamento" dipende dall'implementazione, ma gli oggetti di tipo `char` sono soggetti ai requisiti di allinea-mento meno restrittivi. Come si dirà nel paragrafo A6.8, un puntatore può anche essere convertito in un `void *` e viceversa, senza che queste operazioni ne modifichino il valore.

Un puntatore ad una funzione può essere convertito in un puntatore ad una funzione di un altro tipo. La chiamata della funzione specificata dal puntatore convertito dipende dall'implementazione; tuttavia, se il puntatore convertito viene riportato al suo tipo originario, il risultato è identico al puntatore di partenza.

Un puntatore può essere convertito ad un altro puntatore il cui tipo è uguale a meno della presenza o assenza di qualificatori sul tipo dell'oggetto puntato. Se i qualificatori sono presenti, il nuovo puntatore è equivalente al vecchio a meno delle restrizioni ad esso dovute al nuovo qualificatore. In assenza di quali-ficatori le operazioni sull'oggetto sono governate dai qualificatori presenti nella dichiarazione.

A6.7 VOID

Il valore (inesistente) di un oggetto `void` può non venire utilizzato in alcun modo, ed è anche possibile che ad esso non vengano mai applicate conversioni esplicite né implicite. Poiché un'espressione `void` denota un valore inesistente, essa può essere usata soltanto dove non è richiesto alcun valore, per esempio come operando sinistro dell'operatore virgola (paragrafo A7.18).

Un'espressione può essere convertita al tipo `void` usando l'operatore di cast. Per esempio, per rendere esplicita la volontà di scartare il valore ritornato da una chiamata di funzione usata come espressione.

> Pur appartenendo all'uso comune, il tipo `void` non era presente nella prima edizione di questo libro.

A6.8 Puntatori a VOID

Qualsiasi puntatore può essere convertito nel tipo `void *`, senza alcuna perdita di informazione. Se il risul-tato viene riportato al tipo originario, ciò che si ottiene è il puntatore iniziale. Diversamente da quanto acca-de per la conversione fra puntatori vista nel paragrafo A6.6, che dev'essere sempre esplicita, i puntatori possono essere assegnati a puntatori di tipo `void *`, e possono venire confrontati con essi.

> Quest'interpretazione dei puntatori `void *` è nuova; in precedenza il ruolo dei puntatori generici era ricoperto dal tipo `char *`. Lo standard ANSI favorisce, in particolare, la commistione fra puntatori di tipo `void *` e puntatori ad oggetti all'interno di assegnamenti ed espressioni relazionali, mentre richiede un casting esplicito per le altre commistioni fra puntatori.

A7. Espressioni

La precedenza degli operatori, nelle espressioni, è rispecchiata dall'ordine delle prossime sezioni, in senso decrescente. Quindi, per esempio, le espressioni alle quali ci si riferisce come operandi dell'operatore `+` (descritto nel paragrafo A7.7) sono quelle discusse nei paragrafi da A7.1 ad A7.6. All'interno di ogni sotto-sezione, gli operatori hanno la stessa precedenza. L'associatività destra o sinistra viene specificata di volta in volta. La grammatica incorpora la precedenza e l'associatività degli operatori, e viene riassunta nel para-grafo A7.13.

La precedenza e l'associatività degli operatori sono specificate completamente, ma l'ordine di valutazione delle espressioni, in genere, è indefinito, anche se le sottoespressioni hanno effetti collaterali. Quindi, a me-no che la definizione di un operatore non garantisca un particolare ordine di valutazione dei suoi operandi, l'implementazione è libera di valutarli in un ordine qualsiasi. Tuttavia, ogni operatore combina i valori pro-dotti dai suoi operandi in modo compatibile con l'analisi dell'espressione nella forma in cui essa compare.

> Il comitato ANSI ha deciso, in una fase avanzata dei lavori, di limitare la libertà di riordinamento delle espressioni che coinvolgono operatori matematicamente associativi e commutativi, ma che possono non essere associativi a livello computazionale. In pratica, le variazioni riguardano soltanto i calcoli in floating-point su valori vicini al limite dell'accuratezza, e le situazioni nelle quali si può verificare un overflow.

La gestione dell'overflow, i controlli sulle divisioni e le altre eccezioni che si possono verificare durante la valutazione delle espressioni non sono definite dal linguaggio.

La maggior parte delle attuali implementazioni del C ignora l'overflow nella valutazione di espressioni ed as-segnamenti su variabili intere con segno, ma questo comportamento non è garantito dallo standard.

Il trattamento della divisione per 0, e di tutte le eccezioni sui floating-point, dipende dall'implementazione; talvolta esso è regolabile con l'aiuto di una funzione della libreria non-standard.

A7.1 Generazione di Puntatori

Se il tipo di un'espressione o di una sottoespressione è "vettore di `T`", per un certo tipo `T`, allora il valore dell'espressione è un puntatore al primo elemento del vettore, ed il suo tipo viene alterato in "puntatore a `T`". Questa conversione non ha luogo se l'espressione è l'operando dell'operatore unario `&`, o di `++`, `--` o `sizeof`, oppure se è l'operando sinistro di un operatore di assegnamento o dell'operatore `.` (accesso ad un membro di una struttura). Analogamente, un'espressione del tipo "funzione che ritorna `T`", a meno che non

venga usata come operando dell'operatore `&`, viene convertita nel tipo "puntatore ad una funzione che ritorna `T`". Un'espressione che ha subito una di queste conversioni non è un lvalue.

A7.2 Espressioni Primarie

Le espressioni primarie sono gli identificatori, le costanti, le stringhe e le espressioni racchiuse fra paren-tesi.

```
espressione-primaria
    identificatore
    costante
    stringa
    (espressione)
```

Un identificatore è un'espressione primaria, purché sia stato dichiarato correttamente, secondo quanto de-scritto in precedenza. Il suo tipo è specificato dalla sua dichiarazione. Un identificatore è un lvalue se si rife-risce ad un oggetto (paragrafo A5) e se il suo tipo è aritmetico, struttura, union o procedure.

Una costante è un'espressione primaria. Il suo tipo dipende dal suo formato, discusso nel paragrafo A2.5.

Una stringa letterale è un'espressione primaria. Il suo tipo originario è "vettore di `char`" (per le stringhe di caratteri estesi, il tipo è "vettore di `wchar_t`") ma, in base alle regole illustrate nel paragrafo A7.1, questo ti-po viene di solito modificato in "puntatore a `char`" (`wchar_t`), ed il risultato è un puntatore al primo caratte-re della stringa. Per alcuni inizializzatori (si veda il paragrafo A8.7), questa conversione può non avvenire.

Un'espressione racchiusa fra parentesi è un'espressione primaria, il cui tipo ed il cui valore sono identici a quelli dell'espressione senza parentesi, la presenza delle quali non influisce sul fatto che l'espressione sia un lvalue o meno.

A7.3 Espressioni Postfisse

Nelle espressioni postfisse, gli operatori si raggruppano da sinistra a destra.

```
espressione-postfissa:
    espressione-primaria
    espressione-postfissa [espressione]
    espressione-postfissa (lista-argomenti-espressioneopt)
    espressione-postfissa .identificatore
    espressione-postfissa ->identificatore
    espressione-postfissa ++
    espressione-postfissa -

lista-argomenti-espressione:
    espressione-assegnamento
    lista-argomenti-espressione, espressione-assegnamento
```

A7.3.1 Riferimenti a Vettori

Un'espressione postfissa seguita da un'espressione fra parentesi quadre è un'espressione che denota un riferimento ad un vettore. Una delle due espressioni dev'essere di tipo "puntatore a `T`", dove `T` è un tipo par-ticolare, e l'altra dev'essere di tipo intero; il tipo dell'espressione con indice è `T`. L'espressione `E1[E2]` è identica (per definizione) a `*((E1)+(E2))`. Il paragrafo A8.6.2 approfondisce questo argomento.

A7.3.2 Chiamate di Funzione

Una chiamata di funzione è un'espressione postfissa, detta designatore di funzione, seguita da parentesi tonde che racchiudono una lista (che può anche essere vuota) di espressioni di assegnamento separate da virgole (paragrafo A7.17); queste espressioni costituiscono gli argomenti della funzione. Se l'espressione postfissa è composta da un identificatore che non ha dichiarazione nello scope corrente, l'identificatore

stesso viene implicitamente dichiarato come se, nel blocco più interno contenente la chiamata di funzione, fosse stata data la dichiarazione.

```
extern int  identificatore();
```

L'espressione postfissa (dopo l'applicazione dell'eventuale dichiarazione implicita dev'essere di tipo "punta-tore ad una funzione che ritorna `T`", per un certo tipo `T`, ed il valore della chiamata di funzione è anch'esso di tipo `T`.

> Nella prima edizione, il tipo era semplicemente "funzione" e, per passare ad un puntatore a funzione, era necessario ricorrere ad un operatore `*` esplicito. Lo standard ANSI favorisce l'impiego dei compilatori già esistenti consentendo l'utilizzo della stessa sintassi per le chiamate sia alle funzioni che ai puntatori a funzione. La vecchia sintassi, cioè, è ancora adottabile.

Un'espressione passata con una chiamata di funzione è detta *argomento*; il termine *parametro* indica invece un oggetto in input (od il suo identificatore) ricevuto da una definizione di funzione o descritto in una dichiarazione di funzione. Talvolta, per indicare la stessa distinzione, vengono utilizzati rispettivamente i termini "argomento (parametro) attuale" e "argomento (parametro) formale".

In preparazione ad una chiamata di funzione, di ogni argomento viene fatta una copia; tutto il passaggio di argomenti avviene infatti per valore. Una funzione può modificare i valori dei suoi parametri, che sono delle copie degli argomenti, ma queste modifiche non possono influire sui valori degli argomenti stessi. Tuttavia, è possibile passare anche un puntatore, così che la funzione possa cambiare il valore dell'oggetto puntato.

Esistono due modi di dichiarare una funzione. Secondo la sintassi più recente, i tipi dei parametri sono espliciti e fanno parte del tipo della funzione; una simile dichiarazione è detta anche prototipo della funzio-ne. Nella sintassi originaria, i tipi dei parametri non vengono specificati. La dichiarazione di funzione è di-scussa nei paragrafi A8.6.3 e A10.1.

Se la dichiarazione di una funzione nello scope di una particolare chiamata è di tipo vecchio, agli argomenti vengono applicate le trasformazioni seguenti: su ogni argomento di tipo intero viene eseguita la trasforma-zione degli interi (paragrafo A6.1), ed ogni argomento `float` viene convertito in un `double`. L'effetto della chiamata è indefinito se il numero di argomenti non concorda con il numero di parametri presenti nella defi-nizione della funzione, o se il tipo di un argomento, dopo le trasformazioni, è diverso da quello del parame-tro corrispondente. Il controllo sulla consistenza dei tipi dipende dal fatto che la definizione di funzione sia scritta secondo il vecchio od il nuovo stile. Se essa è nello stile vecchio, il controllo avviene tra il tipo tra-sformato dell'argomento della chiamata, e quello trasformato del parametro; se la definizione è nello stile nuovo, il tipo trasformato dell'argomento dev'essere uguale a quello del parametro.

Se la dichiarazione di una funzione nello scope di una particolare chiamata è di tipo nuovo, gli argomenti vengono convertiti ai tipi dei parametri corrispondenti nel prototipo della funzione. Il numero degli argomenti dev'essere pari a quello dei parametri, a meno che la lista di questi ultimi non termini con la notazione `(, ...)`. In questo caso, il numero degli argomenti dev'essere uguale o maggiore al numero dei parametri; gli argomenti in eccesso rispetto ai parametri con tipo esplicito vengono convertiti in base alle regole descritte nei paragrafi precedenti. Se la definizione di funzione è nello stile vecchio, il tipo di ogni parametro nel pro-totipo visibile alla chiamata dev'essere in accordo con il corrispondente parametro nella definizione, dopo che la definizione del tipo nei parametri è stata sottoposta alla trasformazione degli argomenti.

> Queste regole sono particolarmente complesse perché devono supportare una commistione, da evitare quando possibile, fra vecchio e nuovo stile.

L'ordine di valutazione degli argomenti non è specificato; prendete quindi nota del fatto che i vari compilato-ri possono comportarsi in modo diverso. Tuttavia, gli argomenti ed il designatore di funzione vengono com-pletamente valutati, compresi i loro effetti collaterali, prima di entrare nella funzione. Sono consentite le chiamate ricorsive a qualsiasi funzione.

A7.3.3 Riferimenti a Strutture

Un'espressione postfissa seguita da un punto e da un identificatore è ancora un'espressione postfissa. La espressione che funge da primo operando dev'essere una struttura o una union, e l'identificatore deve indi-carne un membro. Il valore dell'espressione è il membro specificato delle strutture o union, ed il suo tipo

è del membro. L'espressione è un lvalue se la prima espressione lo è, e se il tipo della seconda espressione non è un vettore.

Un'espressione postfissa seguita da una freccia (composta dai segni - e >) e da un identificatore è ancora un'espressione postfissa. L'espressione che funge da primo operando dev'essere un puntatore ad una struttura o ad una union, della quale l'identificatore individua un membro. Il risultato si riferisce al membro specificato della struttura o union puntata dal primo operando, ed il suo tipo è quello del membro; il risultato è un lvalue se il tipo non è un vettore.

Quindi l'espressione `E1->MOS` è uguale a `(*E1).MOS`. Le strutture e le union sono discusse nel paragrafo A8.3.

> Nella prima edizione di questo libro, esisteva già la regola secondo la quale il nome del membro specificato deve appartenere alla struttura o union presente nell'espressione postfissa; tuttavia, una nota ammetteva che questa regola non era ferrea. I compilatori più recenti, ed anche lo standard ANSI, la rafforzano.

A7.3.4 Incremento Postfisso

Un'espressione postfissa seguita dall'operatore `++` o `--` è ancora un'espressione postfissa. Il valore della espressione è il valore dell'operando. Dopo averne registrato il valore, l'operando viene incrementato (`++`) o decrementato (`--`) di uno. L'operando dev'essere un lvalue; per maggiori dettagli relativi agli operatori addi-tivi e di assegnamento si vedano, rispettivamente, i paragrafi A7.7 e A7.17. Il risultato non è un lvalue.

A7.4 Operatori Unari

Le espressioni con operatori unari si raggruppano da sinistra a destra.

```
espressione-unaria:
    espressione-postfissa
    ++espressione-unaria
    --espressione-unaria
    operatore-unario espressione-casting
    sizeof espressione-unaria
    sizeof (nome-di-tipo)

operatore-unario: uno fra
    & * + - ~ !
```

A7.4.1 Operatori Incrementali Prefissi

Un'espressione unaria preceduta da `++` o `--` è ancora un'espressione unaria. L'operando viene incrementato (`++`) o decrementato (`--`) di 1. Il valore dell'espressione è il valore dopo l'incremento (o il decremento). L'operando dev'essere un lvalue; per ulteriori dettagli sugli operatori additivi e di assegna-mento, si vedano i paragrafi A7.7 e A7.17, rispettivamente. Il risultato non è un lvalue.

A7.4.2 Operatore di Indirizzamento

L'operatore unario `&` preleva l'indirizzo del suo operando, che dev'essere un lvalue e non si deve riferire ad un field né ad un oggetto dichiarato `register`; in alternativa, lvalue può essere anche un tipo di funzione. Il risultato è un puntatore all'oggetto o funzione riferito dall'lvalue. Se il tipo dell'operando è `T`, il tipo del ri-sultato è "puntatore a `T`".

A7.4.3 Operatore di Indirezione

L'operatore unario `*` indica l'indirezione, e ritorna l'oggetto o funzione a cui punta l'operando. Esso non è un lvalue se l'operando è un puntatore ad un oggetto di tipo aritmetico, struttura, union o puntatore. Se il tipo dell'espressione è "puntatore a `T`", il tipo del risultato è `T`.

A7.4.4 Operatore Più Unario

L'operando dell'operatore unario `+` dev'essere di tipo aritmetico, ed il risultato è il valore dell'operando. Un operando intero viene sottoposto alla trasformazione degli interi. Il tipo del risultato è il tipo dell'operando trasformato.

L'operatore unario `+` è stato introdotto dallo standard ANSI per simmetria con l'operatore `-` unario.

A7.4.5 Operatore Meno Unario

L'operando dell'operatore unario `-` dev'essere di tipo aritmetico, ed il risultato è il corrispondente valore ne-gativo dell'operando. Un operando intero viene sottoposto alla trasformazione degli interi. Il negativo di una quantità priva di segno viene calcolato sottraendo il valore trasformato dal massimo valore del tipo finale ed aggiungendo uno al risultato; il negativo di zero è zero. Il tipo del risultato è il tipo dell'operando trasfor-mato.

A7.4.6 Operatore di Complemento a Uno

L'operando dell'operatore `-` dev'essere di tipo intero, ed il risultato è il complemento a uno dell'operando. Viene applicata la trasformazione degli interi. Se l'operando è privo di segno, il risultato viene calcolato sot-traendo il suo valore dal massimo valore rappresentabile nel tipo trasformato. Se l'operando ha il segno, il risultato viene calcolato convertendo l'operando nel corrispondente tipo privo di segno, applicandogli l'ope-ratore `~`, e riconvertendo al tipo con segno. Il tipo del risultato è il tipo dell'operando trasformato.

A7.4.7 Operatore di Negazione Logica

L'operando dell'operatore `!` dev'essere di tipo aritmetico o puntatore, ed il risultato è 1 se il valore dell'ope-rando è zero, 0 altrimenti. Il tipo del risultato è `int`.

A7.4.8 Operatore SIZEOF

L'operatore `sizeof` produce il numero di byte richiesti per la memorizzazione di un oggetto del tipo del suo operando. L'operando può essere un'espressione, che non viene valutata, oppure un nome di tipo racchiuso tra parentesi. Quando l'operatore `sizeof` viene applicato ad un `char`, il risultato è 1; quando viene applicato ad un vettore, il risultato è il numero totale di byte del vettore. Se applicato ad una struttura o union, il risultato è il numero di byte dell'oggetto, compresi eventuali arrotondamenti dovuti ad esigenze di allineamento; l'ampiezza di un vettore di *n* elementi è *n* volte la dimensione di un elemento. L'operatore `sizeof` non dovrebbe mai essere applicato ad un operando di tipo funzione, o di tipo incompleto oppure ad un field. Il risultato è una costante intera priva di segno; il suo particolare tipo dipende dall'implementazione. L'header standard `<stddef.h>` (Appendice B) definisce questo tipo come `size_t`.

A7.5 Cast

Un'espressione unaria preceduta da un nome di tipo racchiuso fra parentesi tonde provoca una conversione del valore dell'espressione nel tipo specificato.

```
espressione-casting:
    espressione-unaria
    (nome-tipo) espressione-casting
```

Questa costruzione è detta *cast*. I nomi dei tipi sono descritti nel paragrafo A8.8. Gli effetti delle conversioni sono invece discussi nel paragrafo A6. Un'espressione con un cast non è un lvalue.

A7.6 Operatori Moltiplicativi

Gli operatori moltiplicativi `*`, `/` e `%` si raggruppano da sinistra a destra.

```
espressione-moltiplicativa:
    espressione-casting
    espressione-moltiplicativa * espressione-casting
    espressione-moltiplicativa / espressione-casting
    espressione-moltiplicativa % espressione-casting
```

Gli operandi di `*` e `/` devono essere di tipo aritmetico; gli operandi di `%` devono essere di tipo intero. Su tutti gli operandi vengono applicate le conversioni aritmetiche usuali, che determinano anche il tipo del risultato.

L'operatore binario `*` indica moltiplicazione.

L'operatore binario `/` produce il quoziente, e l'operatore `%` il resto della divisione del suo primo operando per il secondo; se quest'ultimo è 0, il risultato è indefinito. Altrimenti, è sempre vero che `(a/b)*b+a%b` è uguale ad `a`. Se entrambi gli operandi sono non negativi, allora il resto non è negativo e più piccolo del divisore; se non lo sono, viene garantito che il valore assoluto del resto sia inferiore al valore assoluto del divisore.

A7.7 Operatori Additivi

Gli operatori additivi `+` e `-` si raggruppano da sinistra a destra. Se gli operandi sono di tipo aritmetico, su di essi vengono effettuate le conversioni aritmetiche usuali. Per ogni operatore esistono alcune possibilità ag-giuntive sui tipi.

```
espressione-additiva:
    espressione-moltiplicativa
    espressione-additiva + espressione-moltiplicativa
    espressione-additiva - espressione-moltiplicativa
```

Il risultato dell'operatore `+` è la somma degli operandi. Si possono sommare un puntatore ad un elemento di un vettore ed un valore di un qualsiasi tipo intero. Con una moltiplicazione per la dimensione dell'oggetto puntato, l'intero viene convertito in uno spiazzamento di indirizzo. La somma è un puntatore dello stesso ti-po di quello originale, e punta ad un altro elemento del vettore, correttamente spiazzato rispetto all'oggetto originario. Quindi, se `P` è un puntatore ad un oggetto in un vettore, l'espressione `P+1` è un puntatore all'og-getto successivo nel vettore. Se il puntatore risultante dalla somma supera la fine del vettore, il risultato è indefinito a meno che esso non punti alla prima posizione dopo la fine del vettore.

> La gestione dei puntatori immediatamente dopo la fine del vettore è nuova. Essa legittima un idioma comune, da tempo utilizzato per la scansione dei vettori.

Il risultato dell'operatore `-` è la differenza dei suoi operandi. Un valore di qualsiasi tipo intero può essere sottratto da un puntatore, ed a questa operazione si applicano tutte le condizioni illustrate nel caso dell'ad-dizione.

Se si sottraggono fra loro due puntatori ad oggetti dello stesso tipo, il risultato è un intero con segno, che rappresenta la distanza fra i due oggetti puntati; i puntatori ad oggetti adiacenti differiscono di 1. Il tipo del risultato dipende dall'implementazione, ma è definito come `ptrdiff_t` nell'header standard `<stddef.h>`. Il valore è indefinito a meno che i puntatori non puntino ad oggetti appartenenti ad uno stesso vettore; se, però, `P` punta all'ultimo elemento di un vettore, l'espressione `(P+1)-P` ha valore 1.

A7.8 Operatori di Shift

Gli operatori di shift `<<` e `>>` si raggruppano da sinistra a destra. Per entrambi, gli operandi devono essere di tipo intero, e sono soggetti alla trasformazione degli interi. Il tipo del risultato è quello dell'operando di si-nistra trasformato. Il risultato è indefinito se l'operando destro è negativo, oppure se è maggiore o uguale al numero di bit del tipo dell'espressione di sinistra.

```
espressione-shift:
    espressione-additiva
    espressione-shift << espressione-additiva
    espressione-shift >> espressione-additiva
```

Il valore di `E1<<E2` è `E1` (interpretato come sequenza di bit) shiftato a sinistra di `E2` posizioni; in assenza di overflow, questo equivale ad una moltiplicazione per `2^E2`. Il valore di `E1>>E2` è `E1` shiftato a destra di `E2` posizioni. Lo shift a destra equivale ad una divisione per `2^E2` se `E1` è privo di segno o se ha comunque valore non negativo; altrimenti, il risultato è definito dall'implementazione.

A7.9 Operatori Relazionali

Gli operatori relazionali si raggruppano da sinistra a destra, ma ciò non è utile; `a<b<c` viene trattato come `(a<b)<c`, e `a<b` vale sempre 0 o 1.

```
espressione-relazionale
    espressione-shift
    espressione-relazionale < espressione-shift
    espressione-relazionale > espressione-shift
    espressione-relazionale <= espressione-shift
    espressione-relazionale >= espressione-shift
```

Gli operatori `<` (minore di), `>` (maggiore di), `<=` (minore o uguale a) e `>=` (maggiore o uguale a) producono 0 se la relazione specificata è falsa, 1 altrimenti. Il tipo del risultato è `int`. Sugli operandi aritmetici vengono applicate le conversioni aritmetiche usuali. I puntatori ad oggetti dello stesso tipo, ignorando qualsiasi quali-ficatore, possono essere confrontati; il risultato dipende dalla collocazione relativa dei puntatori nello spazio di indirizzamento degli oggetti puntati. Il confronto fra puntatori è definito soltanto per parti di uno stesso og-getto: se due puntatori puntano allo stesso oggetto, essi sono uguali; se puntano a membri di una stessa struttura, i puntatori ai membri dichiarati prima sono i minori; se puntano a membri di una stessa union, essi risultano uguali; se i puntatori si riferiscono ad elementi di un vettore, il confronto equivale ad un confronto degli indici corrispondenti. Se `P` punta all'ultimo membro di un vettore, allora `P+1` risulta maggiore di `P`, an-che se esce dal vettore. In tutti gli altri casi, il confronto fra puntatori è indefinito.

Queste regole diminuiscono leggermente le restrizioni imposte nella prima edizione di questo libro, consentendo il confronto fra puntatori a differenti membri di una struttura o union. Essi legalizzano anche il confronto con un puntatore che cade appena oltre la fine di un vettore.

A7.10 Operatori di Uguaglianza

```
espressione-uguaglianza:
    espressione-relazionale
    espressione-uguaglianza == espressione-relazionale
    espressione-uguaglianza != espressione-relazionale
```

Gli operatori `==` (uguale a) e `!=` (diverso da) sono analoghi agli operatori relazionali, eccetto che per la loro precedenza, più bassa (quindi, `a<b == c<d` vale 1 ogni volta che `a<b` e `c<d` hanno lo stesso valore di verità).

Gli operatori di uguaglianza seguono le stesse regole degli operatori relazionali, ma forniscono possibilità aggiuntive: un puntatore può essere confrontato con un'espressione costante intera con valore 0, o con un puntatore a `void`. Si veda, a questo proposito, il paragrafo A6.6.

A7.11 Operatore AND Bit a Bit

```
espressione-AND:
    espressione-uguaglianza
    espressione-AND & espressione-uguaglianza
```

Vengono applicate le conversioni aritmetiche usuali; il risultato è la funzione AND bit a bit degli operandi. Lo operatore si applica soltanto ad operandi di tipo intero.

A7.12 Operatore OR Esclusivo Bit a Bit

```
espressione-OR-esclusivo:
    espressione-AND
    espressione-OR-esclusivo ^ espressione-AND
```

Vengono applicate le conversioni aritmetiche usuali; il risultato è la funzione OR esclusivo bit a bit degli operandi. L'operatore si applica soltanto ad operandi di tipo intero.

A7.13 Operatore OR Inclusivo Bit a Bit

```
espressione-OR-inclusivo:
    espressione-OR-esclusivo
    espressione-OR-inclusivo | espressione-OR-esclusivo
```

Vengono applicate le conversioni aritmetiche usuali; il risultato è la funzione OR inclusivo bit a bit degli ope-randi. L'operatore si applica soltanto ad operandi di tipo intero.

A7.14 Operatore AND Logico

```
espressione-AND-logico:
    espressione-OR-inclusivo
    espressione-AND-logico && espressione-OR-inclusivo
```

L'operatore `&&` si raggruppa da sinistra a destra. Esso ritorna 1 se entrambi gli operandi sono diversi da ze-ro, 0 altrimenti. Diversamente da `&`, `&&` garantisce una valutazione da sinistra a destra: il primo operando viene valutato, inclusi i suoi effetti collaterali; se esso risulta uguale a zero, il valore dell'intera espressione è 0. Altrimenti, viene valutato l'operando di destra e, se è nullo, il valore dell'intera espressione è 0, altri-menti è 1.

Non è necessario che gli operandi siano dello stesso tipo, purché essi siano tutti di tipo aritmetico o punta-tore. Il risultato è di tipo `int`.

A7.15 Operatore OR Logico

```
espressione-OR-logico:
    espressione-AND-logico
    espressione-OR-logico || espressione-AND-logico
```

L'operatore `||` si raggruppa da sinistra a destra. Esso ritorna 1 se uno dei suoi operandi è diverso da zero, 0 altrimenti. Diversamente da `|`, `||` garantisce una valutazione da sinistra a destra: il primo operando viene valutato, inclusi i suoi effetti collaterali; se esso risulta diverso da zero, il valore dell'intera espressione è 1. Altrimenti, viene valutato l'operando di destra e, se non è nullo, il valore dell'intera espressione è 1, altri-menti è 0.

Non è necessario che gli operandi siano dello stesso tipo, purché essi siano tutti di tipo aritmetico o punta-tore. Il risultato è di tipo `int`.

A7.16 Operatore Condizionale

```
espressione-condizionale:
    espressione-OR-logico
    espressione-OR-logico ? espressione : espressione-condizionale
```

Viene valutata, compresi gli effetti collaterali, la prima espressione; se risulta diversa da 0, il risultato è il valore della seconda espressione, altrimenti quello della terza. Soltanto una delle ultime due espressioni viene valutata. Se esse sono di tipo aritmetico, vengono ridotte ad un tipo comune tramite l'impiego delle conversioni aritmetiche usuali, e questo tipo è anche quello del risultato. Se entrambe le espressioni sono di tipo `void`, o sono strutture o union dello stesso tipo, o puntatori ad oggetti dello stesso tipo, il risultato è del tipo comune. Se un'espressione è un puntatore e l'altra è la costante 0, lo 0 viene convertito nel tipo puntatore, che è anche il tipo del risultato. Se un'espressione è un puntatore a `void` e l'altra è un puntatore di tipo diverso, quest'ultima viene convertita a `void`, che è anche il tipo del risultato.

Nel confronto fra i tipi dei puntatori, tutti i qualificatori di tipo (paragrafo A8.2) degli oggetti puntati sono non significativi, ma il tipo del risultato riguarda i qualificatori di entrambi i rami della condizione.

A7.17 Espressioni di Assegnamento

Esistono diversi operatori di assegnamento; tutti si raggruppano da destra a sinistra.

```
espressione-assegnamento:
    espressione-condizionale
    espressione-unaria operatore-assegnamento espressione-assegnamento

operatore-assegnamento: uno fra
    = *= /= %= += -= <<= >>= &= ^= |=
```

Tutti, come operando sinistro, richiedono un lvalue modificabile; esso non dev'essere un vettore, e non deve avere un tipo incompleto o essere una funzione. Inoltre, il suo tipo non deve possedere il qualificatore `const`; se esso è una struttura o una union, neppure i suoi membri possono essere qualificati come `const`. Il tipo di un'espressione di assegnamento è quello del suo operando di sinistra, ed il suo valore è quello re-gistrato in esso dopo che l'assegnamento ha avuto luogo.

Nell'assegnamento semplice con `=`, il valore dell'espressione sostituisce quello dell'oggetto al quale lvalue si riferisce. Deve sempre essere verificata una delle seguenti condizioni: entrambi gli operandi sono di tipo aritmetico, nel qual caso l'operando di destra viene convertito al tipo di quello di sinistra; entrambi gli opera-tori sono strutture o union dello stesso tipo; un operando è un puntatore, e l'altro è un puntatore a `void`; lo operando sinistro è un puntatore e l'operando destro è un'espressione costante con valore 0; entrambi gli operandi sono puntatori a funzioni od oggetti i cui tipi sono uguali, a meno di una possibile assenza dei qualificatori `const` e `volatile` nell'operando destro.

Un'espressione della forma `E1`*op*`=E2` equivale a `E1=E1`*op*`(E2)`, ad eccezione del fatto che, nella prima for-ma, `E1` viene valutata una sola volta.

A7.18 Operatore Virgola

```
espressione:
    espressione-assegnamento
    espressione, espressione-assegnamento
```

Una coppia di espressioni separate da una virgola viene valutata da sinistra a destra, ed il valore della espressione di sinistra viene scartato. Il tipo ed il valore del risultato sono quelli dell'espressione di destra. Tutti gli effetti collaterali dell'espressione di sinistra vengono valutati prima della valutazione dell'operando destro. Nei contesti nei quali la virgola ha un significato particolare, come per esempio una lista degli argo-menti di una funzione (paragrafo A7.3.2) o una lista di inizializzatori (paragrafo A8.7), l'unità sintattica ri-chiesta è un'espressione di assegnamento, in modo che l'operatore virgola compaia soltanto in un raggrup-pamento; per esempio,

```
f(a, (t=3, t+2), c)
```

ha tre argomenti, il secondo dei quali ha valore 5.

A7.19 Espressioni Costanti

Sintatticamente, un'espressione costante è un'espressione ristretta ad un sottoinsieme di operatori:

```
espressione-costante:
    espressione-condizionale
```

Le espressioni costanti sono necessarie in diversi contesti: dopo un `case`, come limiti di un vettore e come lunghezze dei field, come valore di una costante enumerativa, negli inizializzatori ed in alcune espressioni del preprocessor.

Le espressioni costanti non dovrebbero contenere operatori di assegnamento, incremento o decremento, chiamate di funzione, operatori virgola, se non nell'operatore `sizeof`. Se l'espressione costante dev'essere intera, i suoi operandi devono essere costanti intere, enumerative, carattere e floating; i cast devono speci-ficare un tipo intero, e qualsiasi costante floating dev'essere forzata ad un intero. Questo elimina necessa-riamente le operazioni sui vettori, le indirezioni, l'indirizzamento e l'accesso a membri di strutture (tuttavia, l'operatore `sizeof` può avere operandi di qualsiasi tipo).

Una maggiore libertà viene concessa per le espressioni costanti di inizializzatori; gli operandi possono essere costanti di qualsiasi tipo, e l'operatore unario `&` può venire applicato ad oggetti statici od esterni, ed a vettori statici od esterni indicizzati tramite un'espressione costante. L'operatore unario `&` può venire appli-cato implicitamente utilizzando vettori non indicizzati o funzioni. Gli inizializzatori, una volta valutati, devono produrre una costante oppure l'indirizzo di un oggetto, precedentemente dichiarato statico o esterno, più o meno una costante.

Una libertà minore è concessa sulle espressioni costanti intere che seguono un `#if`; le espressioni conte-nenti `sizeof`, casting e costanti enumerative non sono consentite. Si veda, a questo proposito, il paragrafo A12.5.

A8. Dichiarazioni

Le dichiarazioni specificano l'interpretazione data ad ogni identificatore; non necessariamente esse riserva-no memoria per l'oggetto associato all'identificatore. Le dichiarazioni che lo fanno sono chiamate *definizioni*. Le dichiarazioni hanno la forma:

```
dichiarazione:
    specificatori-dichiarativi lista-dichiaratori-inizialiopt;
```

I dichiaratori nella lista-dichiaratori-iniziali contengono gli identificatori che devono essere dichiarati; gli spe-cificatori-dichiarativi consistono in una sequenza di specificatori di tipo e di classe di memoria.

```
specificatori-dichiarativi:
    specificatore-classe-memoria specificatori-dichiarativiopt
    specificatore-tipo specificatori-dichiarativiopt
    qualificatore-tipo specificatori-dichiarativiopt

lista-dichiaratori-iniziali:
    dichiaratore-iniziale
    lista-dichiaratori-iniziali, dichiaratore-iniziale

dichiaratore-iniziale:
    dichiaratore
    dichiaratore = inizializzatore
```

I dichiaratori verranno discussi nel paragrafo A8.5; essi contengono i nomi che devono essere dichiarati. Una dichiarazione deve avere almeno un dichiaratore, oppure il suo tipo deve dichiarare un tag di una strut-tura o di una union, oppure i membri di un'enumerazione; le dichiarazioni vuote non sono permesse.

A8.1 Specificatori di Classe di Memoria

Gli specificatori di classe di memoria sono:

```
specificatore-classe-memoria:
    auto
    register
    static
    extern
    typedef
```

Il significato di questi specificatori è stato illustrato nel paragrafo A4.

Gli specificatori `auto` e `register` inseriscono nella classe di memoria automatica gli oggetti ai quali si rife-riscono, e dovrebbero essere utilizzati soltanto all'interno delle funzioni. Queste dichiarazioni fungono anche da definizioni, e riservano la memoria per gli oggetti dichiarati. Una dichiarazione `register` equivale ad una `auto`, ma suggerisce che gli oggetti interessati verranno usati spesso. Soltanto pochi oggetti vengono realmente memorizzati nei registri, e solo per alcuni tipi questo è possibile; le restrizioni in proposito dipendono dall'implementazione. In ogni caso, se un oggetto è dichiarato `register`, l'operatore unario `&` non può esservi applicato, né esplicitamente né implicitamente.

La regola secondo la quale è scorretto prelevare l'indirizzo di un oggetto dichiarato `register`, ma che in realtà è di tipo `auto`, è nuova.

Lo specificatore `static` inserisce nella classe di memoria statica gli oggetti ai quali si riferisce, e può veni-re utilizzato sia all'interno che all'esterno delle funzioni. All'interno di una funzione, esso provoca l'allocazio-ne di memoria e funge da definizione; per i suoi effetti all'esterno di una funzione, si veda il paragrafo A11.2.

Una dichiarazione comprendente lo specificatore `extern`, usata all'interno di una funzione, specifica che la memoria per gli oggetti dichiarati è definita altrove; per i suoi effetti all'esterno delle funzioni, si veda il para-grafo A11.2.

Lo specificatore `typedef` non riserva memoria, e viene chiamato specificatore di classe di memoria solo per convenienza sintattica; esso verrà discusso nel paragrafo A8.9.

Una dichiarazione può comprendere al più uno specificatore. Se questo non compare, vengono applicate le seguenti regole: gli oggetti dichiarati all'interno di una funzione vengono considerati `auto`; le funzioni di-chiarate all'interno di altre funzioni vengono considerate `extern`; gli oggetti e le funzioni dichiarati al di fuo-ri di qualsiasi funzione vengono considerati `static`, con linkaggio esterno (si vedano i paragrafi A10 e A11).

A8.2 Specificatori di Tipo

Gli specificatori di tipo sono:

```
specificatore-tipo:
    void
    char
    short
    int
    long
    float
    double
    signed
    unsigned
    specificatore-struttura-o-union
    specificatore-enumerativo
    nome-typedef
```

Insieme ad `int` può essere specificata al più una fra le parole `long` e `short`; il significato non cambia se `int` non viene menzionato. La parola `long` può comparire insieme a `double`. Gli specificatori `signed` e `unsigned` sono mutuamente esclusivi, e possono comparire insieme ad `int` o a qualsiasi sua variante `long` e `short`, oltre che insieme a `char`. Se compaiono da soli, si assume che il tipo sottinteso sia `int`. Lo specificatore `signed` è utile per forzare la presenza del segno negli oggetti di tipo `char`; con gli altri tipi in-teri esso è consentito ma ridondante.

In tutti gli altri casi, una dichiarazione può contenere al più uno specificatore. Se esso viene omesso, si as-sume che venga sottinteso lo specificatore `int`.

Anche i tipi possono venire qualificati, allo scopo di indicare proprietà speciali degli oggetti che vengono dichiarati.

```
qualificatore-tipo:
    const
    volatile
```

I qualificatori di tipo possono comparire a fianco di qualsiasi specificatore. Un oggetto `const` dovrebbe sempre venire inizializzato, ma mai più modificato. Per gli oggetti qualificati `volatile` non esistono se-mantiche indipendenti dall'implementazione.

Le proprietà `const` e `volatile` sono novità introdotte dallo standard ANSI. Lo scopo del qualificatore `const` è quello di indicare oggetti che devono essere posti in memoria a sola lettura consentendo, in alcuni casi, una certa ottimizzazione. Lo

scopo del qualificatore `volatile` è invece quello di eliminare l'ottimizzazione che si potrebbe avere. Per esempio, per una macchina con un input / output mappato in memoria, un puntatore ad un registro di device potrebbe essere qualificato come `volatile`, per impedire che il compilatore rimuova riferimenti, apparentemente ridondanti effettuati tramite questo puntatore. Un compilatore può ignorare questi qualificatori, anche se dovrebbe comunque segnalare eventuali tentativi espliciti di modificare oggetti `const`.

A8.3 Dichiarazioni di Strutture e Union

Una struttura è un oggetto composto da una sequenza di membri di vario tipo. Una union è un oggetto che contiene, in momenti diversi, uno qualsiasi dei suoi membri, anch'essi di vario tipo. Gli specificatori di strut-tura e di union hanno la stessa forma.

```
specificatore-struttura-o-union:
    struttura-o-union identificatore_opt {lista-dichiarazione-struttura}
    struttura-o-union identificatore

struttura-o-union:
    struct
    union
```

Una lista-dichiarazione-struttura è una sequenza di dichiarazioni dei membri della struttura o union:

```
lista-dichiarazione-struttura:
    dichiarazione-struttura
    lista-dichiarazione-struttura dichiarazione-struttura

dichiarazione-struttura:
    lista-specificatori-qualificatori
    lista-dichiaratore-struttura;

lista-specificatori-qualificatori:
    specificatore-tipo lista-specificatori-qualificatori_opt
    qualificatore-tipo lista-specificatori-qualificatori_opt

lista-dichiaratore-struttura:
    dichiaratore-struttura
    lista-dichiaratore-struttura, dichiaratore-struttura
```

Normalmente, un dichiaratore-struttura non è altro che un dichiaratore per un membro di una struttura o union. Un membro di una struttura può consistere anche in un numero specificato di bit. In questo caso il membro è chiamato anche *bit-field*, o semplicemente *field*; la sua lunghezza è separata dal suo nome trami-te un due punti.

```
dichiaratore-struttura:
    dichiaratore
    dichiaratore_opt : espressione-costante
```

Uno specificatore di tipo della forma

```
struttura-o-union identificatore {lista-dichiarazione-struttura}
```

dichiara che l'identificatore è il *tag* della struttura o union specificata dalla lista. Una dichiarazione successi-va, nello stesso scope od in uno più interno, può riferirsi allo stesso tipo utilizzando il tag in uno specificato-re senza lista:

```
struttura-o-union identificatore
```

Se, quando il tag non è dichiarato, compare uno specificatore con il tag stesso ma senza lista, si ha la defi-nizione di un *tipo incompleto*. Gli oggetti con un tipo struttura o union incompleto possono essere utilizzati nei contesti nei quali non è necessaria la loro dimensione, per esempio nelle dichiarazioni (non nelle defini-zioni), per specificare un puntatore o per creare una `typedef`, ma non in tutti gli altri casi. Il tipo diventa completo al momento della successiva occorrenza di quel tag in uno specificatore, contenente una lista di dichiarazioni. Anche negli specificatori aventi una lista, il tipo struttura o union può essere dichiarato

in mo-do incompleto nella lista, e diventa completo soltanto in corrispondenza della parentesi `}` che termina lo specificatore.

Una struttura non può contenere un membro di tipo incompleto. Quindi, è impossibile dichiarare una struttu-ra o union contenente un'istanza in se stessa. Tuttavia, una volta dato un nome alla struttura o union, i tag consentono di creare strutture ricorsive; una struttura o union può contenere un puntatore ad un'istanza di se stessa, perché si possono dichiarare puntatori a tipi incompleti.

Alle dichiarazioni della forma

struttura-o-union identificatore ;

viene applicata una regola molto particolare. Queste dichiarazioni specificano una struttura o union, ma non contengono lista di dichiarazione né dichiaratori. Anche se l'identificatore è il tag di una struttura o union di-chiarata in uno scope più esterno (paragrafo A11.1), questa dichiarazione crea, nello scope corrente, un nuovo tag per un tipo struttura o union incompleto.

> Questa strana regola è stata introdotta dallo standard ANSI. Il suo scopo è quello di consentire il trattamento di strutture mutuamente ricorsive dichiarate in uno scope più interno, ma i cui tag possono essere stati dichiarati in uno scope più esterno.

Uno specificatore di struttura o union con una lista ma senza tag crea un tipo unico; esso può essere riferito direttamente soltanto nella dichiarazione alla quale appartiene.

I nomi dei membri ed i tag non sono in conflitto con alcuna delle altre variabili ordinarie. Uno stesso nome di un membro non può comparire due volte in una stessa struttura o union, ma può essere utilizzato in struttu-re o union diverse.

> Nella prima edizione di questo libro, i nomi dei membri di strutture e union non erano associati a quelli dei loro "genitori". Tuttavia, quest'associazione era già comune nei compilatori che hanno preceduto la definizione dello standard ANSI.

Un membro di una struttura o union che non sia un field può contenere oggetti di qualsiasi tipo. Un field (che non ha bisogno di alcun dichiaratore e, quindi, può anche essere privo di nome) è di tipo `int`, `unsi-gned int` o `signed int`, e viene interpretato come oggetto di tipo intero della lunghezza specificata in bit; il fatto che un field di tipo `int` venga considerato con o senza segno dipende dall'implementazione. I field adiacenti vengono unificati in unità di memoria ed in una direzione dipendenti dall'implementazione. Quan-do un field che ne segue un altro non può essere contenuto in un'unità di memoria già parzialmente occupata, quest'ultima può essere riempita di zeri, oppure il field può venire spezzato. Un field senza nome con ampiezza 0 forza il riempimento, in modo che il field seguente si trovi all'inizio della successiva unità di allocazione.

> Lo standard ANSI rende i field ancora più dipendenti dall'implementazione di quanto non lo fossero nella prima edizione. È consigliabile considerare le regole di gestione dei field come dipendenti dall'implementazione. Le strutture contenenti field possono essere utilizzate in modo portabile per tentare di ridurre la memoria richiesta da una struttura (con un probabile aumento delle istruzioni e del tempo necessario per accedere ai field), oppure in modo non portabile per descrivere la memoria a livello di bit. In quest'ultimo caso, è necessario comprendere a fondo le regole dell'implementazione locale.

I membri di una struttura hanno indirizzi crescenti nell'ordine delle dichiarazioni. Un membro di una struttura che non sia un field è allineato al limite di indirizzamento dipendente dal suo tipo; quindi, nella struttura pos-siamo trovare buchi privi di nome. Se un puntatore ad una struttura viene forzato al tipo di un puntatore al suo primo membro, il risultato si riferisce al primo membro della struttura.

Una union può essere vista come una struttura i cui membri si trovano tutti ad un offset 0, e la cui ampiezza è sufficiente a contenere uno qualsiasi dei suoi membri. In un particolare istante, in una union può essere memorizzato al più uno dei suoi membri. Se un puntatore ad una union viene forzato al tipo di un puntatore ad un membro, il risultato si riferisce a quel membro.

Un semplice esempio di dichiarazione di struttura è il seguente

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
```

```
};
```

che contiene un vettore di 20 caratteri, un intero e due puntatori a strutture dello stesso tipo. Una volta data questa dichiarazione, la dichiarazione

```
struct tnode s, *sp;
```

afferma che `s` è una struttura del tipo descritto ed `sp` è un puntatore ad una struttura di questo tipo. Con queste dichiarazioni, l'espressione

```
sp->count
```

si riferisce al campo `count` della struttura puntata da `sp`;

```
s.left
```

si riferisce al puntatore al sottoalbero sinistro della struttura `s`; e

```
s.right->tword[0]
```

si riferisce al primo carattere del membro `word` del sottoalbero destro di `s`. In generale, un membro di una union non dovrebbe venire controllato, a meno che non sia stato assegnato usando lo stesso membro. Tut-tavia, una garanzia speciale semplifica l'impiego delle union: se una union contiene diverse strutture che condividono una stessa sequenza iniziale, e se in un certo istante contiene una di queste strutture, è possi-bile riferirsi alla parte iniziale comune di una qualsiasi delle strutture contenute. Per esempio, il seguente frammento di codice è corretto:

```
union {
    struct {
        int type;
    }n;
    struct {
        int type;
        int intnode;
    }ni;
    struct {
        int type;
        float floatnode;
    }nf;
}u;
....
u.nf.type=FLOAT;
u.nf.floatnode=3.14;
....
if (u.n.type==FLOAT)
    ...sin(u.nf.floatnode) ...
```

8.4 Enumerazioni

Le enumerazioni sono tipi unici, che assumono valori all'interno di un insieme di costanti chiamate enume-ratori. La forma di uno specificatore enumerativo deriva da quelle delle strutture e delle union.

specificatore-enumerativo:
    enum identificatore$_{opt}$ {*lista-enumeratori*}
    enum *identificatore*

lista-enumeratori:
    enumeratore
    lista-enumeratori, enumeratore

enumeratore:
    identificatore

identificatore = espressione-costante

In una lista di enumeratori, gli identificatori sono dichiarati come costanti di tipo `int`, e possono apparire in tutti i punti nei quali possono apparire le costanti. Se non compaiono enumeratori con `=`, i valori delle co-stanti corrispondenti iniziano dallo 0 ed aumentano di 1 mano a mano che la dichiarazione viene letta da si-nistra a destra. Un enumeratore contenente un `=` assegna all'identificatore associato il valore specificato; gli identificatori successivi continuano la progressione a partire dal valore assegnato.

I nomi degli enumeratori all'interno di uno stesso scope devono essere diversi dagli altri enumeratori e dai nomi delle variabili ordinarie, anche se i valori possono non essere distinti.

Il ruolo dell'identificatore in uno specificatore enumerativo è analogo a quello del tag in uno specificatore di struttura; esso denomina una particolare enumerazione. Le regole per gli specificatori enumerativi con e senza tag e liste sono uguali a quelle per le strutture o union, ad eccezione del fatto che non esistono i tipi enumerativi incompleti; il tag di uno specificatore enumerativo senza una lista di enumeratori deve riferirsi ad uno specificatore avente una lista ed appartenente allo stesso scope.

> Le enumerazioni sono una novità rispetto alla prima edizione di questo libro, anche se fanno parte del linguaggio già da alcuni anni.

A8.5 Dichiaratori

I dichiaratori hanno la seguente sintassi:

```
dichiaratore:
    puntatore_opt dichiaratore-diretto

dichiaratore-diretto:
    identificatore
    (dichiaratore)
    dichiaratore-diretto (espressione-costante_opt)
    dichiaratore-diretto (lista-tipo-parametri)
    dichiaratore-diretto (lista-identificatori_opt)

puntatore:
    * lista-qualificatori-tipo_opt
    * lista-qualificatori-tipo_opt puntatore

lista-qualificatori-tipo:
    qualificatore-tipo
    lista-qualificatori-tipo qualificatore-tipo
```

La struttura dei dichiaratori assomiglia a quella delle espressioni contenenti indirezioni, funzioni e vettori; il raggruppamento è lo stesso.

A8.6 Significato dei Dichiaratori

Una lista di dichiaratori compare dopo una sequenza di specificatori di tipo e di classe di memoria. ogni di-chiaratore dichiara un identificatore principale unico: quello che compare come prima alternativa della pro-duzione *dichiaratore-diretto*. Gli specificatori della classe di memoria si applicano direttamente a questo identificatore, ma il suo tipo dipende dalla forma del suo dichiaratore. Un dichiaratore viene interpretato co-me l'asserzione che, quando il suo identificatore compare in un'espressione avente la stessa forma del di-chiaratore, esso produce un oggetto del tipo specificato.

Considerando soltanto le parti relative al tipo degli specificatori di dichiarazione (paragrafo A8.2) ed un par-ticolare dichiaratore, una dichiarazione ha la forma "`T D`", dove `T` è un tipo e `D` un dichiaratore. Il tipo attri-buito all'identificatore nelle diverse forme del dichiaratore viene descritto induttivamente usando questa no-tazione.

In una dichiarazione `T D`, dove `D` è un identificatore non racchiuso fra parentesi, il tipo dell'identificatore è `T`.

In una dichiarazione `T D`, dove `D` ha la forma

```
( D1 )
```

il tipo dell'identificatore in `D1` è uguale a quello di `D`. Le parentesi non alterano il tipo, ma possono modifica-re i legami di dichiaratori complessi.

A8.6.1 Dichiaratori Puntatore

In una dichiarazione `T D`, dove `D` ha la forma

$$* \text{ lista-qualificatori-tipo}_{opt} \text{ D1}$$

ed il tipo dell'identificatore nella dichiarazione `T D1` è "*modificatore-tipo* `T`", il tipo dell'identificatore `D` è "*mo-dificatore-tipo-lista-qualificatore-tipo* puntatore a `T`". I qualificatori che seguono `*` si applicano al puntatore stesso, e non all'oggetto puntato.

Per esempio, consideriamo la dichiarazione

```
int *ap[];
```

In essa `ap[]` ricopre il ruolo di `D1`; una dichiarazione "`int ap[]`" assegnerebbe ad `ap` il tipo "vettore di `int`", la lista-qualificatori-tipo è vuota ed il modificatore-tipo è "vettore di". Invece, la dichiarazione data as-segna ad `ap` il tipo "vettore di puntatori ad `int`".

Come ulteriori esempi, consideriamo le dichiarazioni

```
int i, *pi, *const cpi=&i;
const int ci=3, *pci;
```

che dichiarano un intero `i` ed un puntatore ad intero, `pi`. Il valore del puntatore costante `cpi` non dovrebbe mai essere modificato; esso punterà sempre alla stessa locazione, anche se il valore al quale si riferisce potrà variare. L'intero `ci` è costante, e non dovrebbe mai essere modificato (anche se può essere inizializ-zato, come nel nostro caso). Il tipo di `pci` è "puntatore a `const int`", e `pci` può puntare ad oggetti diversi, che però non devono mai essere alterati tramite riferimenti a `pci` stesso.

A8.6.2 Dichiaratori Vettore

In una dichiarazione `T D`, nella quale `D` ha la forma

$$\text{D1[espressione-costante}_{opt}\text{]}$$

ed il tipo dell'identificatore nella dichiarazione `T D1` è "*modificatore-tipo* `T`", il tipo dell'identificatore di `D` è "modificatore-tipo vettore di `T`". Se l'espressione-costante è presente, essa dev'essere di tipo intero, ed il suo valore dev'essere positivo. Se l'espressione costante che specifica la dimensione non è presente, il vet-tore ha tipo incompleto.

Un vettore può essere costruito a partire da un tipo aritmetico, da un puntatore, da una struttura o union o da un altro vettore (per generare vettori multidimensionali). Il tipo usato per la costruzione del vettore deve sempre essere completo; esso non può essere un vettore o una struttura di tipo incompleto. Questo implica che, in un vettore multidimensionale, possa essere tralasciata soltanto la prima dimensione. Il tipo di un og-getto di un vettore di tipo incompleto è completato da un'altra dichiarazione, completa, dell'oggetto (para-grafo A10.2), o dalla sua inizializzazione (paragrafo A8.7).

Per esempio,

```
float fa[17], *afp[17];
```

dichiara un vettore di numeri `float` ed un vettori di puntatori a numeri `float`. Inoltre,

```
static int x3d[3][5][7];
```

dichiara un vettore statico tridimensionale, con rango 3*5*7. In particolare, `x3d` è un vettore di tre elementi, ognuno dei quali è un vettore di cinque vettori; ognuno di questi ultimi, a sua volta, è un vettore di sette in-teri. In un'espressione può comparire una qualsiasi delle seguenti forme: `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]`. Le prime tre forme sono espressioni di tipo "vettore", mentre la quarta è di tipo `int`. In particolare, `x2d[i][j]` è un vettore di 7 interi, e `x3d[i]` è un vettore di 5 vettori di 7 interi.

L'operazione di indicizzazione di un vettore è definita in modo che `E1[E2]` sia identico a `*(E1+E2)`. Quin-di, nonostante il suo aspetto asimmetrico, l'indicizzazione è un'operazione commutativa. In base alle regole di conversione che si applicano all'operatore `+` ed ai vettori (paragrafi A6.6, A7.1 e A7.7), se `E1` è un vettore ed `E2` un intero, allora `E1[E2]` si riferisce all'`E2`-esimo elemento di `E1`.

Nell'esempio, `x3d[i][j][k]` equivale a `*(x3d[i][j]+k)`. La prima sottoespressione `x3d[i][j]` viene convertita (paragrafo A7.1) nel tipo "puntatore a vettore di interi"; in base a quando detto nel paragrafo A7.7, la somma implica la moltiplicazione per la dimensione di un intero. Da queste regole segue che i vettori vengono memorizzati per righe (l'ultimo indice varia per primo) e che il primo indice nella dichia-razione aiuta a determinare la quantità di memoria utilizzata da un vettore, ma non gioca alcun altro ruolo nel calcolo degli indici.

A8.6.3 Dichiaratori Funzione

In una dichiarazione di funzione secondo la nuova sintassi `T D`, dove `D` ha la forma

```
D1(lista-tipo-parametri)
```

ed il tipo dell'identificatore nella dichiarazione `T D1` è "*modificatore-tipo* `T`", il tipo dell'identificatore `D` è "*mo-dificatore-tipo* funzione con argomenti *lista-tipo-parametri* che ritorna `T`".

La sintassi dei parametri è la seguente:

```
lista-tipo-parametri:
    lista-parametri
    lista-parametri, ...

lista-parametri:
    dichiarazione-parametro
    lista-parametri, dichiarazione-parametro

dichiarazione-parametro:
    specificatori-dichiarazione dichiaratore
    specificatori-dichiarazione dichiaratore-astrattoopt
```

Nella dichiarazione nel nuovo stile, la lista dei parametri specifica il loro tipo. Come caso speciale, un di-chiaratore di una funzione di nuovo stile senza parametri ha una lista di tipo dei parametri composta unica-mente dalla parola chiave `void`. Se la lista del tipo dei parametri termina con "`, ...`", significa che la fun-zione può accettare più argomenti di quanti non ne siano descritti esplicitamente (paragrafo A7.3.2).

I tipi dei parametri che sono vettori o funzioni vengono trasformati in puntatori, in base alle regole di conver-sione dei parametri (si veda il paragrafo A10.1). L'unico specificatore di classe di memoria consentito nello specificatore di dichiarazione dei parametri è `register`, e questo specificatore viene ignorato a meno che il dichiaratore di funzione non sia una definizione di funzione. Analogamente, se i dichiaratori delle dichiara-zioni dei parametri contengono degli identificatori ed il dichiaratore di funzione non è una definizione, gli identificatori escono immediatamente dallo scope. I dichiaratori astratti, che non si riferiscono ad alcun iden-tificatore, verranno discussi nel paragrafo A8.8.

In una dichiarazione di funzione con sintassi vecchia `T D`, dove `D` ha la forma

```
D1(lista-identificatoriopt)
```

ed il tipo dell'identificatore nella dichiarazione `T D1` è "*modificatore-tipo* `T`", il tipo dell'identificatore `D` è "*mo-dificatore-tipo* funzione con argomenti non specificati che ritorna `T`".

I parametri (se sono presenti) hanno la forma

```
lista-identificatori:
    identificatore
    lista-identificatori, identificatore
```

Nei dichiaratori in vecchio stile, la lista degli identificatori dev'essere assente, a meno che il dichiaratore non funga da definizione di funzione (paragrafo A10.1). Nella dichiarazione non viene fornita alcuna infor-mazione sul tipo dei parametri.

Per esempio, la dichiarazione

```
int  f(), *fpi(), (*pfi)();
```

dichiara una funzione `f` che ritorna un intero, una funzione `fpi` che ritorna un puntatore ad un intero, ed un puntatore `pfi` ad una funzione che ritorna un intero. In nessuna di queste dichiarazioni è specificato il tipo dei parametri: esso sono tutte dichiarazioni in vecchio stile.

Nella dichiarazione nuovo stile

```
int  strcpy(char *dest, const char *source), rand(void);
```

`strcpy` è una funzione con due argomenti, un puntatore a carattere il primo ed un puntatore a carattere co-stante il secondo, e che ritorna un `int`. I nomi dei parametri ne spiegano il significato. La seconda funzione, `rand`, non ha argomenti e ritorna un `int`.

I dichiaratori di funzione con i prototipi dei parametri sono la novità più importante introdotta nel linguaggio dallo standard ANSI. Ri-spetto ai dichiaratori "vecchio stile" della prima edizione, essi hanno il vantaggio di consentire la rilevazione degli errori e la forzatura del tipo degli argomenti al momento delle chiamate di funzione, ma ad un prezzo: lo scompiglio e la confusione dovuti alla loro introduzione, e la necessità di gestire entrambe le forme. Per salvaguardare la compatibilità è stato necessario introdurre degli artifici particolari, quali per esempio lo specificatore `void` esplicito per identificare una funzione nuovo stile senza parametri.

Anche la notazione "`, ...`", per funzioni con numero variabile di argomenti, è nuova e, insieme alle macro contenute nell'header standard `<stdarg.h>`, formalizza un meccanismo ufficialmente proibito ma, di fatto, adottato dalla prima edizione.

Queste notazioni sono state adattate dal linguaggio C++.

A8.7 Inizializzazione

Quando un oggetto viene dichiarato, il suo dichiaratore-iniziale può specificare un valore iniziale da asse-gnare all'identificatore dichiarato. L'inizializzatore è preceduto dal segno `=`, ed è un'espressione oppure una lista di inizializzatori racchiusi fra parentesi graffe. Per una formattazione precisa, la lista dovrebbe sempre terminare con una virgola.

```
inizializzatore:
    espressione-assegnamento
    {lista-inizializzatori}
    {lista-inizializzatori,}

lista-inizializzatori:
    inizializzatore
    lista-inizializzatori, inizializzatore
```

Nell'inizializzatore di un oggetto statico o di un vettore, tutte le espressioni devono essere costanti (paragra-fo A7.19). La stessa regola vale anche nel caso in cui l'inizializzatore è una lista racchiusa fra parentesi graffe, e l'oggetto od il vettore sono dichiarati `auto` o `register`. Se, invece, l'inizializzatore di un oggetto automatico è una singola espressione, essa può non essere costante, purché il suo tipo sia coerente con quello dell'oggetto.

> La prima edizione di questo libro non consentiva l'inizializzazione di strutture, union o vettori automatici. Lo standard ANSI consen-te quest'operazione, ma soltanto su costrutti costanti, a meno che l'inizializzatore possa essere espresso sotto forma di singola espressione.

Un oggetto statico (o i suoi membri, se è una struttura o union) non esplicitamente inizializzato viene inizia-lizzato a 0. Il valore iniziale di un oggetto automatico non esplicitamente inizializzato è indefinito.

L'inizializzatore di un puntatore o di un oggetto di tipo aritmetico è un'espressione semplice, eventualmente racchiusa fra parentesi. L'espressione viene assegnata all'oggetto.

L'inizializzatore di una struttura è un'espressione dello stesso tipo, oppure una lista di inizializzatori per i suoi membri. Se il numero di inizializzatori è inferiore a quello dei membri della struttura, i membri restanti vengono inizializzati a zero. Non dovrebbero mai esistere più inizializzatori che membri. I membri dei field privi di nome sono ignorati e non vengono inizializzati.

L'inizializzatore di un vettore è una lista di inizializzatori per i suoi elementi. Se la dimensione del vettore è sconosciuta, ciò che la determina è il numero degli inizializzatori, ed il tipo del vettore diventa completo. Se il vettore ha ampiezza fissata, il numero degli inizializzatori non deve superare quello degli elementi del vet-tore stesso; se gli inizializzatori sono meno degli elementi, quelli in eccesso fra questi ultimi vengono inizia-lizzati a zero.

Un caso speciale è quello del vettore di caratteri, che può essere inizializzato come una stringa; successivi caratteri della stringa inizializzano successivi elementi del vettore. Analogamente, un letterale con caratteri estesi (paragrafo A2.6) può inizializzare un vettore di tipo `wchar_t`. Se la dimensione del vettore non è no-ta, essa viene determinata dal numero di caratteri della stringa più il carattere di terminazione (`'\0'`); se, invece, l'ampiezza è fissata, il numero di caratteri della stringa, escluso il carattere `'\0'`, dev'essere al massimo pari all'ampiezza stessa.

L'inizializzatore di una union è un'espressione dello stesso tipo, oppure un inizializzatore racchiuso fra pa-rentesi, per il primo elemento della union.

> La prima edizione di questo libro non consentiva l'inizializzazione delle union. La regola del "primo elemento" è grossolana, ma è difficile generalizzarla senza ricorrere ad una nuova sintassi. In compenso, consentendo l'inizializzazione esplicita delle union, que-sta regola dello standard ANSI definisce la semantica di union statiche non esplicitamente inizializzate.

Un *aggregato* è una struttura od un vettore. Se un aggregato contiene, a sua volta, membri di tipo aggrega-to, le regole di inizializzazione vengono applicate ricorsivamente. Nell'inizializzazione, le parentesi graffe possono essere tralasciate osservando il seguente criterio: se l'inizializzatore di un membro di un aggregato che è a sua volta un aggregato inizia con una parentesi graffa sinistra, allora la successiva lista di inizializ-zatori, separata da una virgola, inizializza i membri del sottoaggregato; quindi, la presenza di un numero di inizializzatori superiore a quello dei membri costituisce un errore. Se, tuttavia, l'inizializzatore di un sottoag-gregato non inizia con una parentesi graffa sinistra, allora nella lista vengono considerati soltanto gli ele-menti necessari ad inizializzare i membri del sottoaggregato; gli elementi restanti inizializzano i membri suc-cessivi dell'aggregato al quale il sottoaggregato appartiene.

Per esempio,

```
int x[]={1, 3, 5};
```

dichiara ed inizializza `x` come un vettore monodimensionale con tre elementi, poiché la dimensione non è stata specificata e sono stati elencati tre inizializzatori.

```
float y[4][3]={
    {1, 3, 5},
    {2, 4, 6},
    {3, 5, 7},
};
```

è un'inizializzazione con parentesizzazione completa: 1, 3 e 5 inizializzano la prima riga del vettore, `y[0]`, cioè gli elementi `y[0][0]`, `y[0][1]` e `y[0][2]`. Analogamente vengono inizializzate le righe `y[1]` e `y[2]`. L'inizializzatore termina in anticipo, e quindi gli elementi della riga `y[3]` vengono inizializzati a 0. Lo stesso risultato sarebbe stato prodotto dall'inizializzazione:

```
float y[4][3]={
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

L'inizializzatore di y inizia con una parentesi graffa sinistra, diversamente dall'inizializzatore di y[0]; quin-di, per y[0], vengono presi i primi tre elementi della lista. I successivi tre inizializzano y[1] e gli altri y[2]. Il costrutto

```
float y[4][3]={
    {1}, {2}, {3}, {4}
};
```

inizializza la prima colonna di y (visto come un vettore bidimensionale) e lascia a zero tutti gli altri elementi.

Infine,

```
char msg[]="Errore di sintassi alla linea %s\n";
```

mostra un vettore di caratteri i cui elementi vengono inizializzati con una stringa; la sua dimensione com-prende il carattere nullo di terminazione.

A8.8 Nomi di Tipo

In alcuni contesti (per specificare esplicitamente, con un cast, una conversione di tipo, per dichiarare il tipo dei parametri nei dichiaratori di una funzione e come argomento dell'operatore sizeof) è necessario forni-re il nome di un tipo di dati. Ciò viene fatto usando un *nome di tipo* che, sintatticamente, è una dichiarazione di un oggetto di quel tipo, nella quale il nome dell'oggetto non viene fornito.

```
nome-tipo:
    lista-specificatori-qualificatori dichiaratore-astratto_opt

dichiaratore-astratto:
    puntatore
    puntatore_opt dichiaratore-astratto-diretto

dichiaratore-astratto-diretto:
    (dichiaratore-astratto)
    dichiaratore-astratto-diretto_opt [espressione-costante_opt]
    dichiaratore-astratto-diretto_opt (lista-tipo-parametri_opt)
```

Nel dichiaratore-astratto è possibile identificare la posizione nella quale dovrebbe apparire l'identificatore se la costruzione fosse un dichiaratore all'interno di una dichiarazione. Il tipo nominato è allora uguale al ti-po dell'ipotetico identificatore. Per esempio,

```
int
int *
int *[3]
int (*)[]
int *()
int (*[])(void)
```

indicano, rispettivamente, il tipo "intero", "puntatore ad intero", "vettore di tre puntatori ad intero", "puntatore ad un vettore di un numero imprecisato di interi", "funzione con parametri imprecisati, che ritorna un punta-tore ad intero" e "vettore, di ampiezza imprecisata, di puntatori a funzioni prive di parametri, ognuna delle quali ritorna un intero".

A8.9 TYPEDEF

Le dichiarazioni comprendenti lo specificatore di classe di memoria `typedef` non dichiarano alcun oggetto; esse si limitano a definire degli identificatori per particolari tipi di dati. Questi identificatori vengono chiamati nomi-typedef.

```
nome-typedef:
    identificatore
```

Una dichiarazione `typedef` attribuisce, secondo le regole usuali (paragrafo A8.6), un tipo ad ogni nome compreso nei dichiaratori. Quindi, ogni nome-typedef è sintatticamente identico ad una parola chiave che specifica un tipo.

Per esempio, dopo le dichiarazioni

```
typedef long  Blockno, *Blockptr;
typedef struct {double r, theta; } Complex;
```

le costruzioni

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

sono dichiarazioni legali. Il tipo di `b` è `long`, quello di `bp` è "puntatore a `long`", e quello di `z` è la struttura specificata; `zp`, infine, è un puntatore ad una struttura di questo tipo.

`typedef` non introduce nuovi tipi, ma solo dei sinonimi che potrebbero essere specificati anche in altro mo-do. Nell'esempio, `b` ha lo stesso tipo di un qualsiasi altro oggetto `long`.

I nomi-typedef possono venire ridichiarati in uno scope più interno, ma è indispensabile fornire un insieme non vuoto di specificatori di tipo. Per esempio,

```
extern Blockno;
```

non ridichiara `Blockno`, mentre

```
extern int Blockno;
```

lo fa.

A8.10 Equivalenze di Tipi

Due liste di specificatori di tipo sono equivalenti se contengono lo stesso insieme di specificatori di tipo, te-nendo conto del fatto che alcuni specificatori ne includono altri (per esempio, `long` include `long int`). Le strutture, le union e le enumerazioni con tag diversi sono diverse, ed una struttura, union o enumerazione priva di tag identifica un tipo unico.

Due tipi sono uguali se i loro dichiaratori astratti (paragrafo A8.8), dopo l'espansione dei tipi `typedef` e la cancellazione degli identificatori dei parametri delle funzioni, sono uguali nel senso dell'equivalenza fra le loro liste di specificatori di tipo. Le dimensioni dei vettori ed i tipi dei parametri delle funzioni sono significa-tivi.

A9. Istruzioni

Ad eccezione di casi specifici, che verranno evidenziati, le istruzioni vengono eseguite in sequenza. Le istruzioni vengono eseguite per il loro effetto, e non hanno valore proprio. Esse si suddividono in diversi gruppi.

```
istruzione:
    istruzione-con-label
    istruzione-espressione
    istruzione-composta
```

```
        istruzione-di-selezione
        istruzione-di-iterazione
        istruzione-di-salto
```

A9.1 Istruzioni con Label

Le istruzioni possono avere delle label come prefisso.

```
    istruzione-con-label:
        identificatore : istruzione
        case espressione-costante : istruzione
        default : istruzione
```

Una label è un identificatore che dichiara l'identificatore stesso. L'unico impiego della label è come target di un `goto`. Lo scope dell'identificatore è la funzione corrente. Poiché le label hanno un proprio spazio per il nome, questo non interferisce con gli altri identificatori e non può essere ridichiarato. Si veda, a questo pro-posito, il paragrafo A11.1.

Le label `case` e `default` vengono usate nell'istruzione `switch` (paragrafo A9.4). L'espressione costante associata a `case` dev'essere di tipo intero.

Le label, di per se stesse, non alterano il controllo del flusso.

A9.2 Istruzioni Espressione

La maggior parte delle istruzioni è costituita da istruzioni espressione, che hanno la forma

```
    istruzione-espressione:
        espressione_opt;
```

Molte istruzioni espressione sono assegnamenti o chiamate di funzione. Tutti gli effetti collaterali della espressione vengono completamente valutati prima che venga eseguita l'istruzione successiva. Se l'espres-sione manca, la costruzione è detta istruzione nulla, e viene spesso utilizzata per associare un corpo vuoto ad un'istruzione di iterazione o per collocare una label.

A9.3 Istruzioni Composte

Poiché diverse istruzioni possono essere usate dove, a livello logico, l'operazione sarebbe unica, il C forni-sce l'istruzione composta (detta anche "blocco"). Il corpo di una definizione di funzione è un'istruzione com-posta.

```
    istruzione-composta:
        {lista-dichiarazioni_opt lista-istruzioni_opt}

    lista-dichiarazioni:
        dichiarazione
        lista-dichiarazioni dichiarazione

    lista-istruzioni:
        istruzione
        lista-istruzioni istruzione
```

Se un identificatore della lista-dichiarazioni si trova anche nello scope all'esterno del blocco, la dichiarazio-ne esterna al blocco viene tralasciata (paragrafo A11.1) e ripresa all'uscita del blocco. Un identificatore può essere dichiarato soltanto una volta all'interno di uno stesso blocco. Queste regole si applicano agli identifi-catori collocati nello spazio di nomi (paragrafo A11); identificatori in spazi di nomi diversi sono considerati distinti.

L'inizializzazione degli oggetti automatici viene eseguita ogni volta che si entra nel blocco, e segue l'ordine dei dichiaratori. Se l'ingresso nel blocco non è sequenziale ma avviene a causa di un salto,

quest'inizializ-zazione non viene eseguita. Le inizializzazioni degli oggetti `static` vengono eseguite soltanto una volta, prima dell'inizio dell'esecuzione del programma.

A9.4 Istruzioni di Selezione

Le istruzioni di selezione scelgono uno fra i possibili flussi di controllo.

```
istruzione-di-selezione:
    if (espressione) istruzione
    if (espressione) istruzione else istruzione
    switch (espressione) istruzione
```

In entrambe le forme dell'istruzione `if` l'espressione, che dev'essere di tipo aritmetico o puntatore, viene valutata, con tutti i suoi effetti collaterali, e confrontata con lo 0: se il suo valore è nullo, la sottoistruzione viene eseguita. Nella seconda forma, la seconda sottoistruzione viene eseguita quando l'espressione risulta nulla. L'ambiguità sull'`else` viene risolta associando ogni `else` all'ultimo `if` che ne è privo, incontrato allo stesso livello di nidificazione del blocco.

Il costrutto `switch` provoca il trasferimento del controllo ad una delle sue istruzioni, scelta in base al valore di una particolare espressione, che dev'essere di tipo intero. Tipicamente, la sottoistruzione controllata da uno `switch` è composta. Qualsiasi istruzione all'interno della sottoistruzione può essere etichettata con una o più label `case` (paragrafo A9.1). L'espressione di controllo dello `switch` viene sottoposta alla trasfor-mazione degli interi (paragrafo A6.1), e le costanti associate alle label `case` vengono convertite nel tipo fi-nale dell'espressione. Due `case` associati allo stesso `switch` non possono contenere, dopo la conversio-ne, costanti con valore uguale. Ad ogni `switch` può venire associata al più una label `default`. Gli `switch` possono essere nidificati; ogni label `case` o `default` è associata allo `switch` più interno che la contiene.

Quando l'istruzione `switch` viene eseguita, la sua espressione viene valutata, insieme ai suoi effetti colla-terali, e confrontata con ogni costante associata alle label `case`. Se una di queste costanti risulta uguale al valore dell'espressione, il controllo passa all'istruzione associata a quella label `case`. Se non esistono label `case` con costanti uguali al valore dell'espressione, ed esiste la label `default`, il controllo passa all'istru-zione associata a questa label. Se non esiste neppure la label `default`, non viene eseguita alcuna delle sottoistruzioni dello `switch`.

> Nella prima edizione di questo libro, l'espressione che controllava lo `switch` e le costanti associate alle label `case` dovevano essere esclusivamente di tipo `int`.

A9.5 Istruzioni di Iterazione

Le istruzioni di iterazione definiscono dei cicli.

```
istruzione-di-iterazione:
    while (espressione) istruzione
    do istruzione while (espressione);
    for (espressione_opt; espressione_opt; espressione_opt)
        istruzione
```

Nelle istruzioni `while` e `do`, la sottoistruzione viene eseguita ripetutamente, fino a quando il valore della espressione rimane diverso da zero; l'espressione dev'essere di tipo aritmetico o puntatore. Con l'istruzione `while` il controllo, comprendente tutti gli effetti collaterali derivanti dall'espressione, avviene prima di ogni esecuzione dell'istruzione; con l'istruzione `do`, il controllo avviene invece al termine di ogni esecuzione.

Nell'istruzione `for`, la prima espressione viene valutata una sola volta, e specifica l'inizializzazione del ci-clo. Non esistono restrizioni sul tipo di quest'espressione. La seconda espressione dev'essere di tipo arit-metico o puntatore; essa viene valutata prima di ogni iterazione, ed il ciclo termina quando il suo valore è 0. La terza espressione viene valutata dopo ogni iterazione, e specifica la reinizializzazione del ciclo. Non esi-stono restrizioni sul suo tipo. Gli effetti collaterali di ogni espressione vengono completati immediatamente dopo la sua valutazione. Se la sottoistruzione non contiene alcun `continue`, un'istruzione del tipo

```
for (espressione1; espressione2; espressione3) istruzione
```

equivale al blocco

```
espressione1;
while (espressione2) {
    istruzione
    espressione3;
}
```

Ognuna delle tre espressioni può venire omessa. Omettere la seconda espressione equivale a controllare una costante non nulla.

A9.6 Istruzioni di Salto

Le istruzioni di salto trasferiscono il controllo in modo incondizionato.

```
istruzione-di-salto:
    goto identificatore;
    continue;
    break;
    return espressione_opt;
```

Nell'istruzione `goto`, l'identificatore dev'essere una label (paragrafo A9.1) collocata nella funzione corrente. Il controllo viene trasferito all'istruzione associata alla label.

Un'istruzione `continue` può comparire soltanto all'interno di un'istruzione di iterazione. Essa trasferisce il controllo alla parte di continuazione del più interno ciclo di iterazione che la contiene. Più precisamente, in ognuna delle istruzioni

```
while (....) {
    ....
  contin: ;
}
do {
    ....
  contin: ;
} while (....);

for (....) {
    ....
  contin: ;
}
```

un'istruzione `continue` non contenuta in un'istruzione di iterazione più interna equivale ad un `goto con-tin`.

Un'istruzione `break` può comparire soltanto in un'istruzione di iterazione o in uno `switch`, e termina l'ese-cuzione dell'istruzione di iterazione (o `switch`) più interna che la contiene; il controllo passa all'istruzione che segue quella appena terminata.

Una funzione restituisce il controllo dell'esecuzione al chiamante tramite l'istruzione `return`. Quando que-st'ultima è seguita da un'espressione, il valore restituito al chiamante è quello dell'espressione stessa. La espressione viene convertita, come se si trattasse di un assegnamento, al tipo fornito dalla funzione nella quale compare.

Terminare una funzione equivale ad un ritorno senza espressione. In entrambi i casi, il valore restituito è in-definito.

A10. Dichiarazioni Esterne

L'unità di input fornita dal compilatore C viene chiamata unità di traduzione; essa consiste in una sequenza di dichiarazioni esterne, che sono o dichiarazioni o definizioni di funzioni.

```
unità-traduzione:
    dichiarazione-esterna
    unità-traduzione dichiarazione-esterna

dichiarazione-esterna:
    definizione-funzione
    dichiarazione
```

Lo scope delle dichiarazioni esterne si estende fino al termine dell'unità di traduzione nella quale sono di-chiarate, così come l'effetto delle dichiarazioni all'interno dei blocchi si estende fino al termine del blocco. La sintassi delle dichiarazioni esterne è uguale a quella delle altre dichiarazioni, ad eccezione del fatto che il codice per le funzioni può essere scritto soltanto a questo livello.

A10.1 Definizione di Funzioni

Le definizioni di funzioni hanno la forma:

```
definizione-funzione:
    specificatori-dichiarazione_opt dichiaratore
        lista-dichiarazioni_opt istruzione-composta
```

Negli specificatori di dichiarazione, gli unici specificatori di classe di memoria consentiti sono `extern` e `static`; per le differenze fra i due, si veda il paragrafo A11.2.

Una funzione può restituire un valore di tipo aritmetico, una struttura, una union, un puntatore o un `void`, ma non una funzione o un vettore. In una dichiarazione di funzione, il dichiaratore deve specificare esplici-tamente che l'identificatore dichiarato è di tipo funzione; cioè, il dichiaratore deve contenere una delle forme (paragrafo A8.6.3)

```
dichiaratore-diretto (lista-tipo-parametri)
dichiaratore-diretto (lista-identificatori_opt)
```

dove dichiaratore-diretto è un identificatore, eventualmente racchiuso fra parentesi. In particolare, esso non può definire il tipo della funzione usando `typedef`.

Nella prima forma, la definizione riguarda una funzione nuovo stile, i cui parametri, insieme ai loro tipi, sono dichiarati nella lista-tipo-parametri; la lista-dichiarazioni che segue il dichiaratore di funzione dev'essere as-sente. A meno che la lista del tipo dei parametri contenga soltanto l'identificatore `void`, che indica una fun-zione priva di parametri, ogni dichiaratore deve contenere un identificatore. Se la lista termina con "`, ...`" la funzione può essere chiamata con un numero di argomenti superiore a quello dei parametri; per riferirsi agli argomenti in eccedenza, dev'essere utilizzato il meccanismo della macro `va_arg`, definito nell'header standard `<stdarg.h>` e descritto nell'Appendice B. Le funzioni con numero variabile di argomenti devono avere nominato esplicitamente almeno un parametro.

Nella seconda forma, la definizione riguarda una funzione vecchio stile: la lista di identificatori elenca i pa-rametri, ai quali la lista di dichiarazione attribuisce il tipo. Se per un parametro non viene data alcuna di-chiarazione, si assume che il suo tipo sia `int`. La lista di dichiarazioni deve dichiarare soltanto i parametri elencati nella lista degli identificatori non è consentita e l'unico specificatore di classe di memoria utilizzabi-le è `register`.

In entrambe le forme di definizione delle funzioni, i parametri vengono considerati dichiarati subito dopo lo inizio dell'istruzione composta che forma il corpo della funzione dove, quindi, gli stessi identificatori non possono essere ridichiarati (anche se, come qualsiasi altro identificatore, potrebbero essere ridichiarati nei blocchi più interni). Se il tipo di un parametro è dichiarato "vettore di *tipo*", la dichiarazione viene letta come "puntatore a *tipo*"; analogamente, se il parametro è dichiarato di tipo "funzione che ritorna *tipo*", la dichiara-zione viene letta come "puntatore a funzione che ritorna *tipo*". Durante la chiamata ad una funzione, gli ar-gomenti vengono convertiti secondo necessità ed assegnati ai parametri; si veda il paragrafo A7.3.2.

Le definizioni di funzione del nuovo stile sono state introdotte dallo standard ANSI. Esiste anche una piccola variazione nei dettagli di trasformazione; la prima edizione specificava che le dichiarazioni di parametri `float` venivano lette come `double`. La differen-za diventa notevole quando in una funzione viene generato un puntatore ad un parametro.

Un esempio completo di definizione di funzione nel nuovo stile è

```
int  max(int a, int b, int c)
{
    int m;

    m=(a>b)?a:b;
    return (m>c)?m:c;
}
```

In questo esempio, `int` è lo specificatore di dichiarazione; `max(int a, int b, int c)` è il dichiaratore di funzione, e `{....}` è il blocco che costituisce il codice della funzione. La corrispondente definizione di fun-zione nel vecchio stile è

```
int  max(a, b, c)
int a, b, c;
{
    /* .... */
}
```

dove il dichiaratore è `int  max(a, b, c)`, e `int a, b, c;` è la lista di dichiarazione dei parametri.

A10.2 Dichiarazioni Esterne

Le dichiarazioni esterne specificano le caratteristiche degli oggetti, delle funzioni e degli altri identificatori. Il termine "esterne" si riferisce alla loro collocazione, esterna alle funzioni, e non è direttamente connesso alla parola chiave `extern`; la classe di memoria di un oggetto dichiarato esternamente può non essere esplici-ta, oppure può essere specificata come `extern` o `static`.

Purché siano consistenti nel tipo e nel linkaggio, e purché non ci sia al più una definizione per l'identificato-re, nella stessa unità di traduzione possono esistere più dichiarazioni esterne per uno stesso identificatore.

Due dichiarazioni di un oggetto o funzione sono considerate consistenti in base alle regole descritte nel pa-ragrafo A8.10. Inoltre, se le dichiarazioni sono diverse perché un tipo è una struttura, una union o un'enu-merazione di tipo incompleto (paragrafo A8.3) e l'altro è il tipo corrispondente completato e con lo stesso tag, i tipi sono considerati consistenti. Ancora, se un tipo è un vettore di tipo incompleto (paragrafo A8.6.2) e l'altro è un tipo vettore completato, i tipi sono considerati consistenti. Infine, se un tipo specifica una fun-zione vecchio stile e l'altro una funzione nuovo stile che, se non fosse per questo, sarebbe identica alla pri-ma, i tipi sono considerati consistenti.

Se la prima dichiarazione esterna di un oggetto o funzione comprende lo specificatore `static`, l'identifica-tore ha *linkaggio interno*; altrimenti ha *linkaggio esterno*. Il linkaggio viene discusso nel paragrafo A11.2.

Una dichiarazione esterna per un oggetto è una definizione se possiede un inizializzatore. Una dichiarazio-ne di un oggetto esterna e priva di inizializzazione, e che non contiene lo specificatore `extern`, è una *defi-nizione sperimentale*. Se in un'unità di traduzione compare una definizione di un oggetto, tutte le definizioni sperimentali relative a tale oggetto vengono considerate ridondanti. Se nell'unità di traduzione non compa-iono definizioni dell'oggetto, tutte le definizioni sperimentali relative ad esso diventano un'unica definizione con inizializzatore 0.

Ogni oggetto deve avere una ed una sola definizione. Per oggetti con linkaggio interno, questa regola si ap-plica separatamente ad ogni unità di traduzione, perché tali oggetti sono unici rispetto ad un'unità di tradu-zione. Per oggetti con linkaggio esterno, la regola si applica all'intero programma.

Anche se la regola della definizione unica era formulata in modo leggermente diverso nella prima edizione di questo libro, il suo effetto era identico a quello descritto qui. Alcune implementazioni ne diminuiscono la rigidità generalizzando la nozione di definizio-ne sperimentale. Nella formulazione alternativa, diffusa nei sistemi UNIX e riconosciuta come estensione dello Standard, tutte le definizioni sperimentali relative ad un oggetto linkato esternamente, in tutte le unità di traduzione del programma, vengono conside-rate insieme, invece che separatamente per ogni unità. Se in qualche punto del programma è presente una definizione, allora le definizioni sperimentali diventano semplicemente delle dichiarazioni ma, se non compare alcuna definizione, esse diventano una unica definizione con inizializzatore 0.

A11. Scope e Link

Un programma non ha bisogno di essere compilato tutto contemporaneamente: il testo sorgente può essere mantenuto in più file contenenti diverse unità di traduzione, e dalle librerie possono essere caricate routine precompilate. La comunicazione tra le funzioni di un programma può avvenire sia tramite le chiamate che attraverso la manipolazione di dati esterni.

Quindi, è necessario considerare due tipi di scope: il primo è lo *scope lessicale* di un identificatore, che è la regione di testo del programma all'interno della quale le caratteristiche dell'identificatore sono conosciute; il secondo scope è, invece, quello associato ad oggetti e funzioni con linkaggio esterno, che determina le connessioni tra identificatori residenti in unità di traduzione compilate separatamente.

A11.1 Scope Lessicale

Gli identificatori cadono all'interno di spazi di nomi che non interferiscono l'uno con l'altro; lo stesso identifi-catore può essere utilizzato per scopi diversi, anche nello stesso scope, se tali usi avvengono in spazi di nomi distinti. Queste classi sono: oggetti, funzioni, nomi-typedef e costanti `enum`; label, tag di strutture, union ed enumerazioni; membri individuali di strutture o union.

Queste regole differiscono, per alcuni aspetti, da quelle date nella prima edizione di questo manuale. In precedenza le label non avevano un loro spazio separato; i tag delle strutture e delle union avevano ciascuno un loro spazio separato così come, in alcune implementazioni, i tag delle enumerazioni; l'inserimento di diversi tipi di tag in uno stesso spazio è una restrizione nuova. La più importante differenza rispetto alla prima edizione è che ogni struttura crea uno spazio separato per i suoi membri, in modo che lo stesso nome possa comparire in strutture diverse. Questa regola, comunque, fa parte dell'uso comune già da alcuni anni.

Lo scope lessicale di un oggetto o funzione in una dichiarazione esterna inizia al termine del suo dichiarato-re e si estende fino al termine dell'unità di traduzione nella quale l'identificatore compare. Lo scope di un parametro di una definizione comincia all'inizio del blocco che definisce la funzione, e si estende per tutta la funzione; lo scope di un parametro in una dichiarazione di funzione termina alla fine del dichiaratore. Lo scope di un identificatore dichiarato all'inizio di un blocco si estende dalla fine del dichiaratore alla fine del blocco. Lo scope di una label è l'intera funzione nella quale compare. Lo scope di un tag di una struttura, di una union o di un'enumerazione o di una costante enumerativa va dalla sua comparsa nello specificatore di tipo al termine dell'unità di traduzione (per le dichiarazioni a livello esterno), oppure al termine del blocco (per dichiarazioni all'interno di una funzione).

Se un identificatore viene esplicitamente dichiarato all'inizio di un blocco, compreso il blocco di definizione di una funzione, qualsiasi dichiarazione dell'identificatore all'esterno del blocco viene sospesa fino alla ter-minazione del blocco stesso.

A11.2 Link

All'interno di un'unità di traduzione, tutte le dichiarazioni dello stesso identificatore di oggetto o funzione con linkaggio interno si riferiscono alla stessa cosa, e l'oggetto o funzione è unico in quell'unità di traduzione. Tutte le dichiarazioni dello stesso identificatore di oggetto o funzione con linkaggio esterno si riferiscono al-la stessa cosa, e l'oggetto o funzione viene condiviso dall'intero programma.

Come si è detto nel paragrafo A10.2, la prima dichiarazione esterna di un identificatore gli fornisce un lin-kaggio interno se contiene lo specificatore `static`, esterno altrimenti. Se una dichiarazione di un identifica-tore all'interno di un blocco non contiene lo specificatore `extern`, allora l'identificatore non ha linkaggio ed è visibile solo a quella funzione. Se, invece, la dichiarazione contiene lo specificatore `extern`, e nello sco-pe che circonda il blocco è attiva una dichiarazione esterna per quell'identificatore, allora quest'ultimo ha linkaggio uguale a quello della dichiarazione esterna, e si riferisce allo stesso oggetto o funzione; se, infine, non è visibile alcuna dichiarazione esterna, l'identificatore ha linkaggio esterno.

A12. Preprocessing

Un preprocessor effettua sostituzioni delle macro, compilazioni condizionali ed inclusioni di file. Le linee che iniziano con il carattere #, eventualmente preceduto da spazi bianchi, comunicano con il preprocessor. La sintassi di queste linee è indipendente dal resto del linguaggio; esse possono comparire ovunque ed hanno un effetto che permane (indipendentemente dallo scope) fino al termine dell'unità di traduzione. La fine del-le linee è significativa; ogni linea viene analizzata individualmente (ma il paragrafo A12.2 spiega come unire due linee). Per il preprocessor, un token è un qualsiasi token del linguaggio, od una sequenza di caratteri che descrive un nome di file in una direttiva `#include` (paragrafo A12.4); inoltre, ogni carattere non defini-to altrimenti viene considerato un token. Tuttavia, nelle linee di preprocessor l'effetto di caratteri di spaziatu-ra diversi dal tab orizzontale è indefinito.

L'attività di preprocessing ha luogo in più fasi logicamente successive, che in alcune particolari implementa-zioni possono essere condensate.

1. Dapprima, le sequenze triplici (descritte nel paragrafo A12.1) vengono sostituite con il loro equi-valente. Se l'ambiente gestito dal sistema operativo lo richiede, tra le linee del file sorgente vengono inseriti dei caratteri di new line.
2. Ogni occorrenza del carattere backslash seguito da un new line viene cancellata; le linee vengono cioè unite (paragrafo A12.2).
3. Il programma viene separato in token suddivisi da spazi bianchi; i commenti vengono sostituiti con un singolo spazio. A questo punto vengono eseguite le direttive di preprocessing e vengono espanse le macro (paragrafi da A12.3 a A12.10).
4. Le sequenze di escape nelle costanti carattere e nelle stringhe letterali (paragrafi A2.5.2 e A2.6) ven-gono sostituite con i loro equivalenti; quindi, le stringhe letterali adiacenti vengono concatenate.
5. Il risultato viene tradotto, quindi linkato insieme agli altri programmi e librerie, unendo i programmi ed i dati necessari, e connettendo i riferimenti a funzioni ed oggetti esterni alle rispettive definizioni.

A12.1 Sequenze Triplici

Il set di caratteri dei programmi C è contenuto in un codice ASCII a sette bit, ma è un sovrainsieme dell'ISO 646-1983 Invariant Code Set. Per consentire la rappresentazione dei programmi nel set ridotto, tutte le oc-correnze delle seguenti sequenze triplici vengono sostituite con il corrispondente carattere singolo. Questa sostituzione avviene prima di qualsiasi altri valutazione.

```
??=   #
??/   \
??'   ^
??(   [
??)   ]
??!   |
??<   {
??>   }
??-   ~
```

Non avvengono altre sostituzioni.

Le sequenze triplici sono una novità introdotta dallo standard ANSI.

A12.2 Unione di Linee

Le linee che terminano con un carattere backslash \ vengono unite cancellando tale carattere ed il new line che lo segue. Questo avviene prima della separazione dei token.

A12.3 Definizione ed Espansione delle Macro

Una linea di controllo della forma

`#define` *`identificatore sequenza-di-token`*

fa in modo che il preprocessor sostituisca tutte le istanze dell'identificatore con la data sequenza di token; gli spazi bianchi che precedono e seguono la sequenza vengono scartati. La presenza di una seconda `#define` per lo stesso identificatore costituisce un errore, a meno che la seconda sequenza non sia identi-ca alla prima, considerando il fatto che tutte le separazioni costituite da spazi bianchi sono considerate equivalenti.

Una linea della forma

```
#define identificatore(lista-idenfiticatori) sequenza-di-token
```

senza spazi fra il primo identificatore e la parentesi tonda sinistra, definisce una macro con i parametri dati nella lista degli identificatori. Come nel primo caso, gli spazi bianchi che precedono e seguono la sequenza di token vengono scartati, e la macro può essere ridefinita soltanto con una definizione in cui il numero dei parametri e la sequenza di token sono uguali.

Una linea di controllo della forma

```
#undef identificatore
```

annulla la definizione di un identificatore del preprocessor. L'applicazione di `#undef` ad un identificatore sconosciuto non costituisce un errore.

Quando una macro è stata definita nel secondo modo, le successive istanze testuali dell'identificatore della macro, seguite da spazi bianchi opzionali, da `{`, una sequenza di token separati da virgole e da `}`, costitui-scono una chiamata della macro stessa. Gli argomenti della chiamata sono i token separati dalle virgole; le virgole tra apici o protette da parentesi nidificate non separano gli argomenti. Il numero di argomenti della chiamata dev'essere uguale a quello dei parametri nella definizione. Dopo aver isolato gli argomenti, gli spazi bianchi che li precedono e li seguono vengono scartati. Quindi, la sequenza di token risultante da ogni argomento viene sostituita ad ogni occorrenza, che non sia tra apici, dell'identificatore della macro nel-la sequenza di token che costituisce la macro stessa. A meno che il parametro nella sequenza non sia pre-ceduto da `#`, o preceduto o seguito da `##`, i token che costituiscono gli argomenti vengono esaminati, e se necessario espansi, prima dell'inserimento.

Due operatori particolari influiscono sul procedimento di sostituzione. Se un'occorrenza di un parametro, in una sequenza di token nel corpo della macro, è immediatamente preceduta da `#`, intorno al parametro ven-gono collocati i doppi apici ("), e sia il carattere `#` che l'identificatore del parametro vengono sostituiti dallo argomento tra apici. Il carattere `\` viene inserito prima di ogni carattere " che circonda o si trova all'interno di una stringa letterale o costante carattere che compare nell'argomento.

L'altro operatore speciale è `##`. Se la definizione della sequenza di token per uno qualsiasi dei due tipi di macro contiene un operatore `##`, allora, subito dopo la sostituzione dei parametri, ogni `##` viene cancellato insieme agli spazi che lo precedono e lo seguono, in modo che due token adiacenti vengano concatenati e ne formino uno nuovo, singolo. L'effetto è indefinito se ciò che viene prodotto è un token non valido, o se il risultato dipende dall'ordine di valutazione degli operatori `##`. Inoltre, `##` non può comparire all'inizio od alla fine di una sequenza di token.

In entrambi i tipi di macro, la sequenza di token da sostituire viene scandita ripetutamente per identificatori definiti più volte. Tuttavia, una volta che un identificatore è stato sostituito in una data espansione, esso non viene più sostituito, anche se ricompare in una scansione successiva.

Anche se il valore finale dell'espansione di una macro inizia con `#`, esso non deve essere interpretato come una direttiva di preprocessor.

> I dettagli del processo di espansione delle macro sono definiti più precisamente nello standard ANSI che non nella prima edizione. La variazione più importante è l'aggiunta degli operatori `#` e `##`, che rendono possibili il quoting e la concatenazione. Alcune delle nuove regole, in particolare quelle relative alla concatenazione, sono piuttosto strane (si vedano gli esempi che seguono).

Per esempio, questa definizione può essere utilizzata per le costanti esplicite:

```
#define  TABSIZE  100
int  table[TABSIZE];
```

La definizione

```
#define  ABSDIFF(a,b)  ((a)>(b)?(a)-(b):(b)-(a))
```

definisce una macro che restituisce il valore assoluto della differenza dei suoi argomenti. Diversamente da quanto accadrebbe con una funzione che eseguisse le stesse operazioni, gli argomenti ed il valore restituito dalla macro possono essere di qualsiasi tipo aritmetico, oltre che dei puntatori. Notiamo poi che gli argo-menti, che potrebbero anche avere effetti collaterali, vengono valutati due volte, una per il controllo e l'altra per la generazione del valore.

Data la definizione

```
#define  tempfile(dir)  #dir "/%s"
```

la chiamata `tempfile(/usr/tmp)` produce le stringhe

```
"/usr/tmp" "/%s"
```

che, successivamente, verranno concatenate in una singola stringa. Dopo la definizione

```
#define  cat(x,y)  x ## y
```

la chiamata `cat(var,123)` produce `var123`. Invece, la chiamata `cat(cat(1,2),3)` è indefinita: la pre-senza dell'operatore `##` impedisce agli argomenti della chiamata esterna di essere espansi. Questa chiama-ta, quindi, produce la stringa di token

```
cat(1,2)3
```

e `)3` (la concatenazione dell'ultimo token del primo argomento con il primo token del secondo) non è un to-ken legale. Se si introduce un secondo livello di definizione di macro,

```
#define  xcat(x,y)  cat(x,y)
```

le cose si svolgono diversamente; `xcat(xcat(1,2),3)` produce `123`, perché l'espansione di `xcat` non coinvolge direttamente l'operatore `##`.

Analogamente, `ABSDIFF(ABSDIFF(a,b),c)` produce il risultato atteso, completamente espanso.

A12.4 Inclusione di File

Una linea di controllo della forma

```
#include  <nomefile>
```

provoca la sostituzione di questa linea con l'intero contenuto del file *nomefile*. I caratteri del nome *nomefile* non devono comprendere `>` o new line, e l'effetto è indefinito se contengono un carattere qualsiasi fra `"`, `'`, `\` o `/*`. Il file specificato viene cercato in una serie di luoghi dipendente dall'implementazione.

In modo simile, una linea della forma

```
#include  "nomefile"
```

cerca dapprima un file associato al file sorgente originale (una frase, questa, deliberatamente dipendente dall'implementazione) e, se questa ricerca fallisce, il comportamento diviene identico a quello assunto nella prima forma. L'effetto dell'impiego di `'`, `\` o `/*` nel nome del file rimane indefinito, ma il carattere `>` è con-sentito.

Infine, una direttiva del tipo

```
#include  sequenza-di-token
```

che non corrisponde a nessuna delle forme precedenti viene interpretata espandendo la sequenza di token come se si trattasse di normale testo; il risultato di quest'operazione dev'essere una delle due forme <....> o "....", che viene trattata secondo le modalità già descritte.

Le direttive #include possono essere nidificate.

A12.5 Compilazione Condizionale

Le parti di un programma possono essere compilate in modo condizionale, in base alla seguente sintassi schematica.

```
condizione-di-preprocessor:
    linea-if testo parti-elif parte-else_opt #endif

linea-if:
    #if espressione-costante
    #ifdef identificatore
    #ifndef identificatore

parti-elif:
    linea-elif testo
    parti-elif_opt

linea-elif:
    #elif espressione-costante

parte-else:
    linea-else testo

linea-else:
    #else
```

Ognuna delle direttive (linea-if, linea-elif, linea-else e #endif) compare da sola su una linea. Le espressio-ni costanti nella linea #if e nelle successive linee #elif vengono valutate nell'ordine, fino a quando non viene trovata un'espressione con valore non nullo; il testo che segue una linea con valore zero viene scar-tato. Il testo che segue invece una linea con valore non nullo viene trattato normalmente. In questo contesto il termine "testo" si riferisce a qualsiasi tipo di materiale, incluse le linee di preprocessor, che non appartie-ne alla struttura condizionale; tale materiale potrebbe anche essere nullo. Una volta individuata una linea #if o #elif con valore non nullo, e dopo il trattamento del suo testo, le successive linee #elif e #else vengono scartate, insieme al loro testo. Se tutte le espressioni sono nulle, ed esiste una linea #else, il te-sto che la segue viene trattato normalmente. Il testo controllato da rami inattivi della struttura condizionale viene ignorato, se non per il controllo dell'annidamento delle condizioni.

L'espressione costante nelle linee #if e #elif è soggetta alle ordinarie sostituzioni sulle macro. Inoltre, ogni espressione del tipo

```
defined identificatore
```

oppure

```
defined (identificatore)
```

viene sostituita con 1L se l'identificatore è definito nel preprocessor, con 0L altrimenti. Tutti gli identificatori che rimangono dopo l'espansione delle macro vengono sostituiti con 0L. Infine, ogni costante intera viene considerata come se avesse il suffisso L, in modo che tutte le operazioni aritmetiche si svolgano su oggetti long o unsigned long.

L'espressione costante risultante (paragrafo A7.19) ha delle restrizioni: essa dev'essere di tipo intero, e non può contenere sizeof, cast o costanti enumerative.

Le linee di controllo

```
#ifdef identificatore
#ifndef identificatore
```

sono equivalenti a

```
#if defined identificatore
#if !defined identificatore
```

rispettivamente.

> `#elif` è una direttiva nuova rispetto alla prima edizione, anche se era già da tempo disponibile su alcuni preprocessor. Anche lo operatore di preprocessor `defined` è nuovo.

A12.6 Controllo di Linea

Per il beneficio di altri preprocessor che generano programmi C, una linea avente una delle due forme

```
#line costante "nomefile"
#line costante
```

fa in modo che il compilatore, a scopo di diagnostica, creda che il numero di linea della prossima linea di codice sorgente sia dato dalla costante decimale intera e che il corrente file di input sia quello nominato dal-l'identificatore. Se il nome del file è assente, il nome memorizzato non cambia. Le macro nella linea vengo-no espanse prima che essa venga interpretata.

A12.7 Generazione di Errori

Una linea di preprocessor avente il seguente formato

#error sequenza-di-token$_{opt}$

fa in modo che il processore scriva un messaggio diagnostico contenente la sequenza di token.

A12.8 Pragma

Una linea di controllo del tipo

#pragma sequenza-di-token$_{opt}$

fa in modo che il processore esegua un'operazione dipendente dall'implementazione. Una direttiva `#prag-ma` che non viene riconosciuta viene ignorata.

A12.9 Direttiva Nulla

Una linea di preprocessor del tipo

```
#
```

non ha alcun effetto.

A12.10 Nomi Predefiniti

Per produrre informazioni speciali, sono definiti ed espansi alcuni identificatori particolari. Essi, insieme allo operatore di preprocessor `defined`, non dovrebbero mai essere ridefiniti o cancellati (tramite `#undef`).

`__LINE__`	Costante decimale contenente il numero della corrente linea di codice sorgente.
`__FILE__`	Stringa letterale contenente il nome del file del quale è in corso la compilazione
`__DATE__`	Stringa letterale contenente la data della compilazione, nel formato "`Mmm dd yyyy`".
`__TIME__`	Stringa letterale contenente l'ora della compilazione, nel formato "`hh:mm:ss`".

__STDC__ La costante 1. Questo identificatore è definito a 1 soltanto nelle implementazioni con-formi allo standard.

Le direttive `#error` e `#pragma` sono state introdotte dallo standard ANSI; le macro di preprocessor predefinite sono nuove, ma alcune di esse erano già da tempo disponibili su alcune implementazioni.

A13. Grammatica

Quella che segue è una ricapitolazione della grammatica data nel corso dei paragrafi di quest'appendice. Il suo contenuto è uguale a quello esposto in precedenza, ma è ordinato in modo diverso.

La grammatica ha dei simboli terminali non definiti: *costante-intera*, *costante-carattere*, *costante-floating*, *identificatore*, *stringa* e *costante-enumerativa*; lo stile `typewriter` viene utilizzato per parole e simboli ter-minali dati letteralmente. Questa grammatica può essere trasformata meccanicamente in un input per un analizzatore sintattico automatico. Oltre ad aggiungere tutto ciò che è necessario per individuare le alterna-tive in una produzione, è necessario espandere le costruzioni "uno fra", e (seguendo le regole dell'analizza-tore sintattico) duplicare ogni produzione contenente il simbolo "*opt*", una volta con il simbolo ed una volta senza. Con un'ulteriore modifica, che consiste nel cancellare la produzione *nome-typedef*: *identificatore* e trasformare *nome-typedef* in un simbolo terminale, questa grammatica viene accettata dal parser YACC. Rispetto ad esso, questa grammatica contiene un unico conflitto, dovuto all'ambiguità generata dal costrut-to `if-else`.

```
unità-traduzione:
    dichiarazione-esterna
    unità-traduzione dichiarazione-esterna

dichiarazione-esterna:
    definizione-funzione
    dichiarazione

definizione-funzione:
    specificatori-dichiarazione_opt dichiaratore
lista-dichiarazioni_opt istruzione-composta

dichiarazione:
    specificatori-dichiarazione_opt lista-dichiaratori-iniziali;

lista-dichiarazioni:
    dichiarazione
    lista-dichiarazioni dichiarazione

specificatori-dichiarazione:
    specificatore-classe-memoria specificatori-dichiarazione_opt
    specificatore-tipo specificatori-dichiarazione_opt
    qualificatore-tipo specificatori-dichiarazione_opt

specificatore-classe-memoria: uno fra
    auto  register  static  extern  typedef

specificatore-tipo: uno fra
    void  char  short  int  long  float  double  signed
    unsigned specificatore-struttura-o-union
    specificatore-enumerativo nome-typedef

qualificatore-tipo: uno fra
    const  volatile

specificatore-struttura-o-union:
    struttura-o-union identificatore_opt {lista-dichiarazione-struttura}
    struttura-o-union identificatore

struttura-o-union: uno fra
```

```
    struct  union

lista-dichiarazione-struttura:
    dichiarazione-struttura
    lista-dichiarazione-struttura dichiarazione-struttura

lista-dichiaratori-iniziali:
    dichiaratore-iniziale
    lista-dichiaratori-iniziali, dichiaratore-iniziale

dichiaratore-iniziale:
    dichiaratore
    dichiaratore = inizializzatore

dichiarazione-struttura:
    lista-specificatori-qualificatori
lista-dichiaratore-struttura;

lista-specificatori-qualificatori:
    specificatore-tipo lista-specificatori-qualificatori_opt
    qualificatore-tipo lista-specificatori-qualificatori_opt

lista-dichiaratore-struttura:
    dichiaratore-struttura
    lista-dichiaratore-struttura, dichiaratore-struttura

dichiaratore-struttura:
    dichiaratore
    dichiaratore_opt: espressione-costante

specificatore-enumerativo:
    enum identificatore_opt {lista-enumeratori}
    enum identificatore

lista-enumeratori:
    enumeratore
    lista-enumeratori, enumeratore

enumeratore:
    identificatore
    identificatore = espressione-costante

dichiaratore:
    puntatore_opt dichiaratore-diretto

dichiaratore-diretto:
    identificatore
    (dichiaratore)
    dichiaratore-diretto [espressione-costante_opt]
    dichiaratore-diretto (lista-tipo-parametri)
    dichiaratore-diretto (lista-identificatori_opt)

puntatore:
    * lista-qualificatori-tipo_opt
    * lista-qualificatori-tipo_opt puntatore

lista-qualificatori-tipo:
    qualificatore-tipo
    lista-qualificatori-tipo qualificatore-tipo

lista-tipo-parametri:
    lista-parametri
    lista-parametri, ...
```

```
lista-parametri:
    dichiarazione-parametro
    lista-parametri, dichiarazione-parametro

dichiarazione-parametro:
    specificatori-dichiarazione dichiaratore
    specificatori-dichiarazione dichiaratore-astrattoopt

lista-identificatori:
    identificatore
    lista-identificatori, identificatore

inizializzatore:
    espressione-assegnamento
    {lista-inizializzatori}
    {lista-inizializzatori, }

lista-inizializzatori:
    inizializzatore
    lista-inizializzatori, inizializzatore

nome-tipo:
    lista-specificatori-qualificatori dichiaratore-astrattoopt

dichiaratore-astratto:
    puntatore
    puntatoreopt dichiaratore-astratto-diretto

dichiaratore-astratto-diretto:
    (dichiaratore-astratto)
    dichiaratore-astratto-direttoopt [espressione-costanteopt]
    dichiaratore-astratto-direttoopt (lista-tipo-parametriopt)

nome-typedef:
    identificatore

istruzione:
    istruzione-con-label
    istruzione-espressione
    istruzione-composta
    istruzione-di-selezione
    istruzione-di-iterazione
    istruzione-di-salto

istruzione-con-label:
    identificatore: istruzione
    case espressione-costante: istruzione
    default: istruzione

istruzione-espressione:
    espressioneopt;

istruzione-composta:
    {lista-dichiarazioniopt lista-istruzioniopt}

lista-istruzioni:
    istruzione
    lista-istruzioni istruzione

istruzione-di-selezione:
    if (espressione) istruzione
    if (espressione) istruzione else istruzione
```

```
    switch (espressione) istruzione

istruzione-di-iterazione:
    while (espressione) istruzione
    do istruzione while (espressione);
    for (espressione_opt; espressione_opt; espressione_opt)
        istruzione

istruzione-di-salto:
    goto identificatore;
    continue;
    break;
    return espressione_opt;

espressione:
    espressione-assegnamento
    espressione, espressione-assegnamento

espressione-assegnamento:
    espressione-condizionale
    espressione-unaria operatore-assegnamento espressione-assegnamento

operatore-assegnamento: uno fra
    = *= /= %= += -= <<= >>= &= ^= |=

espressione-condizionale:
    espressione-OR-logico
    espressione-OR-logico ? espressione : espressione-condizionale

espressione-costante:
    espressione-condizionale

espressione-OR-logico:
    espressione-AND-logico
    espressione-OR-logico || espressione-AND-logico

espressione-AND-logico:
    espressione-OR-inclusivo
    espressione-AND-logico && espressione-OR-inclusivo

espressione-OR-inclusivo:
    espressione-OR-esclusivo
    espressione-OR-inclusivo | espressione-OR-esclusivo

espressione-OR-esclusivo:
    espressione-AND
    espressione-OR-esclusivo ^ espressione-AND

espressione-AND:
    espressione-uguaglianza
    espressione-AND & espressione-uguaglianza

espressione-uguaglianza:
    espressione-relazionale
    espressione-uguaglianza == espressione-relazionale
    espressione-uguaglianza != espressione-relazionale

espressione-relazionale:
    espressione-shift
    espressione-relazionale < espressione-shift
    espressione-relazionale > espressione-shift
    espressione-relazionale <= espressione-shift
    espressione-relazionale >= espressione-shift
```

```
espressione-shift
    espressione-additiva
    espressione-shift << espressione-additiva
    espressione-shift >> espressione-additiva

espressione-additiva:
    espressione-moltiplicativa
    espressione-additiva + espressione-moltiplicativa
    espressione-additiva - espressione-moltiplicativa

espressione-moltiplicativa:
    espressione-casting
    espressione-moltiplicativa * espressione-casting
    espressione-moltiplicativa / espressione-casting
    espressione-moltiplicativa % espressione-casting

espressione-casting:
    espressione-unaria
    (nome-tipo) espressione-casting

espressione-unaria:
    espressione-postfissa
    ++espressione-unaria
    --espressione-unaria
    operatore-unario espressione-casting
    sizeof espressione-unaria
    sizeof (nome-di-tipo)

operatore-unario: uno fra
    & * + - ~ !

espressione-postfissa:
    espressione-primaria
    espressione-postfissa [espressione]
    espressione-postfissa (lista-argomenti-espressione_opt)
    espressione-postfissa .identificatore
    espressione-postfissa ->identificatore
    espressione-postfissa ++
    espressione-postfissa -

espressione-primaria:
    identificatore
    costante
    stringa
    (espressione)

lista-argomenti-espressione:
    espressione-assegnamento
    lista-argomenti-espressione, espressione-assegnamento

costante:
    costante-intera
    costante-carattere
    costante-floating
    costante-enumerativa
```

La seguente grammatica per il preprocessor riassume la struttura del controllo delle linee, ma non è adatta ad un'analisi automatica. Essa comprende il simbolo *testo*, che significa testo normale del programma, linee di controllo non condizionali del preprocessor, oppure costruzioni complete del preprocessor.

```
linea-di-controllo:
```

```
    #define identificatore sequenza-di-token
    #define identificatore(identificatore, ....,
                  identificatore) sequenza-di-token
    #undef identificatore
    #include <nomefile>
    #include “nomefile”
    #include sequenza-di-token
    #line costante “nomefile”
    #line costante
    #error sequenza-di-token opt
    #pragma sequenza-di-token opt
    #
    condizione-di-preprocessor

condizione-di-preprocessor:
    linea-if testo parti-elif parte-else opt #endif

linea-if:
    #if espressione-costante
    #ifdef identificatore
    #ifndef identificatore

parti-elif:
    linea-elif testo
    parti-elif opt

linea-elif:
    #elif espressione-costante

parte-else:
    linea-else testo

linea-else:
    #else
```

APPENDICE B

LIBRERIA STANDARD

In questa appendice viene riassunta la libreria definita dallo standard ANSI. La libreria standard non appar-tiene propriamente al linguaggio C, ma qualsiasi ambiente che supporta il C fornisce le dichiarazioni di fun-zione e le definizioni dei tipi e delle macro proprie di questa libreria. Abbiamo omesso un piccolo insieme di funzioni di utilità limitata o che, comunque, sono ben rappresentate da altre; abbiamo trascurato il tratta-mento dei caratteri su più byte; inoltre, abbiamo tralasciato la discussione degli aspetti locali, cioè di tutte le proprietà che dipendono dal linguaggio locale, dalla nazionalità o dalla cultura.

Le funzioni, i tipi e le macro della libreria standard sono dichiarate negli *header*.

```
<assert.h>      <float.h>       <math.h>        <stdarg.h>
<ctype.h>       <limits.h>      <setjmp.h>      <stddef.h>
<errno.h>       <locale.h>      <signal.h>      <stdio.h>
<stdlib.h>      <string.h>      <time.h>
```

Ad un header si può accedere utilizzando l'istruzione

```
#include <header>
```

Gli header possono essere inclusi in qualsiasi ordine ed un qualsiasi numero di volte. Un header dev'essere incluso al di fuori di qualunque dichiarazione o definizione esterna, e prima che venga utilizzata una qual-siasi entità definita in esso. Un header può non essere un file sorgente.

Gli identificatori esterni che iniziano con il carattere underscore sono riservati alla libreria, così come tutti gli identificatori nei quali il primo underscore è seguito da un secondo o da una lettera maiuscola.

B1. Input ed Output: STDIO.H

Le funzioni, i tipi e le macro definiti in `<stdio.h>` costituiscono circa un terzo della libreria.

Uno *stream* è una sorgente o una destinazione di dati che possono essere associati ad un disco oppure ad altre periferiche. La libreria supporta stream di testo e binari, anche se essi, su alcuni sistemi (per esempio UNIX) sono identici. Uno stream di testo è una sequenza di linee; ogni linea possiede zero o più caratteri ed è terminata dal carattere `'\n'`. Un particolare ambiente può avere bisogno di convertire uno stream di testo da una rappresentazione ad un'altra (per esempio, tradurre ogni carattere `'\n'` in un return o in un line feed). Uno stream binario è una sequenza di byte non processati che registrano dati interni, e garantisce che una scrittura ed una successiva lettura sullo stesso sistema producano gli stessi caratteri.

Uno stream viene connesso ad un file o ad un device tramite un'operazione di *apertura*; la connessione vie-ne introdotta *chiudendo* lo stream. L'apertura di un file fornisce un puntatore ad un oggetto di tipo `FILE`, che contiene tutte le informazioni necessarie per la gestione dello stream. Dove non si presenteranno ambigui-tà, noi utilizzeremo indifferentemente i termini "puntatori a file" e "stream".

Quando un programma entra in esecuzione, i tre stream `stdin`, `stdout` e `stderr` sono già stati aperti.

B1.1 Operazioni sui File

Le operazioni sui file sono gestite dalle seguenti funzioni. Il tipo `size_t` è il tipo intero privo di segno pro-dotto dall'operatore `sizeof`.

`FILE *fopen(const char *filename, const char *mode)`

`fopen` apre il file desiderato, e restituisce uno stream, o `NULL` se l'operazione fallisce. I valori possi-bili di `mode` comprendono:

`"r"` apre in lettura il file di testo
`"w"` crea in scrittura il file di testo; se esistono, scarta i vecchi contenuti

"a" appende; apre o crea il file di testo e lo predispone alla scrittura in coda al contenuto già eventualmente esistente
"r+" apre il file di testo in aggiornamento (cioè in lettura e scrittura)
"w+" crea il file di testo in aggiornamento; se esistono, scarta i vecchi contenuti
"a+" appende; apre o crea il file di testo in aggiornamento, scrivendo in fondo

Le modalità di aggiornamento consentono di leggere e scrivere lo stesso file; tra una lettura ed una scrittura, e viceversa, è necessario eseguire una `fflush` oppure una funzione di riposizionamento sul file. Se il parametro `mode` contiene anche la lettera `b`, come in "`rb`" o in "`w+b`", il file è binario. I nomi dei file possono avere al massimo `FILENAME_MAX` caratteri. Possono venire aperti al più `FOPEN_MAX` file contem-poraneamente.

`FILE *freopen(const char *filename, const char *mode, FILE *stream)`

`freopen` apre il file desiderato e lo associa allo stream. Essa restituisce `stream`, oppure `NULL` se la operazione fallisce. `freopen` viene di solito utilizzata per cambiare i file associati a `stdin`, `stdout`, e `stderr`.

`int fflush(FILE *stream)`

Su uno stream di output, `fflush` provoca la scrittura di dati bufferizzati ma non ancora scritti; su uno stream di input, l'effetto è indefinito. Essa restituisce `EOF` in caso di errore di scrittura, e zero altrimenti. `fflush(NULL)` scarica tutti i buffer associati ai file di output.

`int fclose(FILE *stream)`

`fclose` provoca la scrittura di qualsiasi dato non ancora scritto relativo a `stream`, scarta qualsiasi input bufferizzato ma non ancora letto, libera automaticamente ogni buffer allocato e chiude lo stream. Essa restituisce `EOF` se rileva qualche errore, e zero altrimenti.

`int remove(const char *filename)`

`remove` rimuove il file desiderato, in modo che ogni successivo tentativo di aprirlo fallisca. Se l'ope-razione fallisce, essa restituisce un valore diverso da zero.

`int rename(const char *oldname, const char *newname)`

`rename` cambia il nome di un file; in caso di errore essa restituisce un valore non nullo.

`FILE *tmpfile(void)`

`tmpfile` crea un file temporaneo con modalità "`wb+`"; tale file verrà rimosso automaticamente al momento della chiusura esplicita oppure in caso di terminazione corretta del programma. `tmpfile` restituisce uno stream, oppure `NULL` se non riesce a creare il file.

`char *tmpnam(char s[L_tmpnam])`

`tmpnam(NULL)` crea una stringa diversa da qualsiasi nome di file esistente, e ritorna un puntatore ad un vettore statico interno. `tmpnam(s)` memorizza la stringa in `s` e restituisce `s` come valore di ritorno. `tmpnam` genera un nome diverso ogni volta che viene invocata; durante l'esecuzione di un programma, viene garantita la generazione di al più `TMP_MAX` nomi diversi. Notate che `tmpnam` crea un nome, e non un file.

`int setvbuf(FILE *stream, char *buf, int mode, size_t size)`

`setvbuf` controlla la bufferizzazione di un particolare stream; essa dev'essere invocata prima di una lettura, una scrittura o qualsiasi altra operazione. La modalità `_IOFBF` provoca una bufferizza-zione piena, `_IOLBF` una bufferizzazione a linee dei file di testo, `_IONBF` elimina la bufferizzazione. Se `buf` è diverso da `NULL`, esso verrà utilizzato come buffer; in caso contrario, il buffer verrà alloca-to automaticamente. `size` determina la dimensione del buffer. `setvbuf` restituisce un valore diverso da zero in caso di errore.

`void setbuf(FILE *stream, char *buf)`

Se `buf` è uguale a `NULL`, la bufferizzazione dello stream viene eliminata. Altrimenti, `setbuf` equivale alla chiamata `(void) setvbuf(stream, buf, _IOFBF, BUFSIZ)`.

B1.2 Output Formattato

Le funzioni `printf` forniscono le conversioni per l'output formattato.

`int fprintf(FILE *stream, const char *format, ...)`

`fprintf` converte e scrive l'output su `stream`, sotto il controllo del parametro `format`. Il valore restituito è il numero di caratteri scritti, oppure un valore negativo in caso di errore.

La stringa di formato contiene due tipi di oggetti: caratteri normali, che vengono copiato sullo stream di out-put, e specifiche di conversione, ognuna delle quali provoca la conversione e la stampa del successivo ar-gomento di `fprintf`. Ogni specifica di conversione inizia con il carattere `%` e termina con il carattere di conversione. Tra questi due caratteri si possono inserire, nell'ordine:

- Flag (in un ordine qualsiasi), che modificano la specifica di conversione: `-`, che specifica un incolon-namento destro dell'argomento all'interno del suo campo.

  `+`, che specifica che il numero verrà sempre stampato insieme al suo segno.

  spazio: se il primo carattere non è un segno, verrà stampato uno spazio.

  `0`: per le conversioni numeriche, specifica un incolonnamento in cui il numero è eventualmente pre-ceduto da un opportuno numero di zeri.

  `#`, che specifica un formato alternativo dell'output. Con `o`, la prima cifra è uno zero. Con `x` o `X`, un risultato non nullo viene preceduto con il suffisso `0x` o `0X`. Con `e`, `E`, `f`, `F`, `g` e `G` l'output ha sempre il punto decimale; con `g` e `G`, eventuali zeri non significativi non vengono rimossi.

- Un numero che specifica un'ampiezza minima del campo. L'argomento convertito viene stampato in un campo di ampiezza almeno pari a quella specificata. Se l'argomento convertito ha meno caratteri della ampiezza del campo, esso viene incolonnato a destra (o a sinistra, se è stato richiesto un inco-lonnamento sinistro) in modo da raggiungere l'ampiezza opportuna. Il carattere usato per incolonna-re è, di norma, lo spazio bianco, ma diventa `0` se è stato fornito il flag opportuno.
- Un punto, che separa l'ampiezza del campo dalla precisione.
- Un numero, la precisione, che specifica il massimo numero di caratteri stampabili da una stringa, op-pure il numero di cifre da stampare dopo il punto decimale in presenza dei flag `e`, `E` o `f`, oppure il numero di cifre significative per i flag `g` e `G`, oppure il numero minimo di cifre da stampare per un in-tero (per raggiungere l'ampiezza desiderata verranno aggiunti degli zeri iniziali).
- Un modificatore di lunghezza `h` oppure `l` (lettera elle) oppure `L`; "`h`" indica che l'argomento corri-spondente dev'essere stampato come `short` o come `unsigned short`; "`l`" indica che esso deve essere stampato come `long` o come `unsigned long`; "`L`" indica che l'argomento è un `long dou-ble`.

L'ampiezza o la precisione, o anche entrambe, possono essere indicate con `*`, nel qual caso il loro valore viene calcolato convertendo l'argomento (o gli argomenti) successivo, che dev'essere di tipo `int`.

I caratteri di conversione ed il loro significato sono illustrati nella Tabella B.1. Se il carattere che segue `%` non è un carattere di conversione, il comportamento è indefinito.

`int printf(const char *format, ...)`

`printf(....)` equivale a `fprintf(stdout, ....)`.

Tabella B.1 *Conversioni di PRINTF.*

CARATTERE	TIPO DELL'ARGOMENTO; CONVERTITO IN
`d, i`	`int`; notazione decimale con segno.
`o`	`int`; notazione ottale priva di segno (senza zero iniziale).
`x, X`	`int`; notazione esadecimale priva di segno (senza `0x` o `0X` iniziale), stampato usando `abcdef` o `ABCDEF` per 10, ..., 15.
`U`	`int`; notazione decimale privo di segno.
`c`	`int`; carattere singolo, dopo la conversione a `unsigned char`.

s	char *; stampa caratteri dalla stringa fino al raggiungimento di '\0' o della precisione.
f	double; [-]*m.dddddd*, dove il numero delle *d* è dato dalla precisione (il default è 6).
e, E	double; [-]*m.dddddd*e±*xx* oppure [-]*m.dddddd*E±*xx*, dove il numero delle *d* è dato dalla precisione (il default è 6). La precisione 0 sopprime il punto decimale.
g, G	double; usa %e o %E se l'esponente è minore di -4 o maggiore o uguale alla precisione; altrimenti usa %f. Gli zeri superflui non vengono stampati.
p	void *; puntatore (rappresentazione dipendente dall'implementazione).
n	int *; il numero dei caratteri stampati fino a questo momento in questa chiamata a printf viene *scritto nell'argomento*. Nessun argomento viene convertito.
%	non converte alcun argomento; stampa un %.

`int sprintf(char *s, const char *format, ...)`

sprintf è uguale a printf, ma stampa il suo output nella stringa s, terminata con il carattere '\0'. s dev'essere sufficientemente ampia da potere contenere il risultato. Il valore restituito non include il carattere di terminazione.

```
vprintf(const char *format, va_list arg)
vfprintf(FILE *stream, const char *format, va_list arg)
vsprintf(char *s, const char *format, va_list arg)
```

Le funzioni vprintf, vfprintf e vsprintf equivalgono alle funzioni del tipo printf corrispon-denti, se si esclude il fatto che la lista di argomenti di lunghezza variabile è sostituita da arg, che è stato inizializzato con la macro va_start e con chiamate a va_arg. A questo proposito si veda la discussione relativa all'header <stdarg.h>, al paragrafo B7.

B1.3 Input Formattato

Le funzioni scanf gestiscono la conversione di input formattato.

`int fscanf(FILE *stream, const char *format, ...)`

fscanf legge da stream sotto il controllo di format, ed assegna i valori convertiti ai suoi succes-sivi argomenti, *ognuno dei quali dev'essere un puntatore*. Essa termina quando ha esaurito format. fscanf restituisce EOF in caso di raggiungimento della fine del file o di errore prima di qualsiasi conversione; altrimenti, restituisce il numero di elementi di input convertiti ed assegnati.

Normalmente la stringa di formato contiene le specifiche di conversione, che vengono utilizzate per inter-pretare l'input. La stringa di formato contiene:

- Spazi bianchi, che vengono ignorati.
- Caratteri normali (escluso %, che ci si aspetta concordino con i successivi caratteri non bianchi dello stream di input.
- Specifiche di conversione, composte da %, un carattere * opzionale di soppressione dell'assegna-mento, un numero opzionale che specifica un'ampiezza massima del campo, un carattere opzionale h, l o L che indica la dimensione del target ed un carattere di conversione.

Una specifica di conversione determina la conversione del successivo elemento in input. Di norma, il risul-tato viene memorizzato nella variabile puntata dall'argomento corrispondente. Se è presente il carattere di soppressione dell'assegnamento, come in %*s, l'elemento in input viene scartato. Un elemento in input è definito come una stringa priva di caratteri di spaziatura (tra i quali è compreso il new line); esso si estende fino al successivo spazio o fino al raggiungimento della dimensione del campo, se questa è specificata. Questo implica che scanf, per trovare il suo input, possa scandire più linee, poiché i new line sono consi-derati degli spazi (i caratteri di spaziatura sono gli spazi, i new line, i return, i tab orizzontali e verticali ed i salti pagina).

Il carattere di conversione indica l'interpretazione del campo in input. L'argomento corrispondente dev'esse-re un puntatore. I caratteri di conversione consentiti sono elencati nella Tabella B.2.

Tabella B.2 *Conversioni di SCANF*

CARATTERE	**DATI IN INPUT; TIPO DELL'ARGOMENTO**

d	intero decimale; int *.
I	intero; int *. L'intero può essere in ottale (preceduto da uno 0) oppure in esadecimale (preceduto da 0x o 0X).
O	intero ottale (preceduto o meno dallo 0); int *.
X	intero esadecimale (preceduto o meno da 0x o 0X); int *.
C	Caratteri; char *. I prossimi caratteri in input (1 per default) vengono inseriti nella posi-zione indicata; vengono considerati anche i caratteri di spaziatura; per leggere il prossi-mo carattere non bianco, usate %1s.
s	stringa di caratteri (non racchiusa fra apici); char *, che punta ad un vettore di caratteri sufficientemente grande da contenere la stringa ed uno '\0' di terminazione, che verrà aggiunto automaticamente.
e, f, g	Numero floating-point con segno, punto decimale ed esponente opzionali; float *.
P	Valore del puntatore, così come viene stampato da printf("%p"); void *.
N	Scrive nell'argomento il numero di caratteri letti fino a questo momento in questa chia-mata; int *. Non viene letto alcun input. Il numero degli elementi letti non viene incre-mentato.
[...]	Trova la più lunga stringa non vuota di caratteri di input corrispondenti all'insieme rac-chiuso fra le parentesi; char *. Aggiunge il carattere '\0'. L'insieme []...] comprende il carattere].
[^...]	Trova la più lunga stringa non vuota di caratteri di input *non* corrispondenti all'insieme racchiuso fra le parentesi; char *. Aggiunge il carattere '\0'. L'insieme [^]...] com-prende il carattere].
%	Carattere %; non viene effettuato alcun assegnamento.

I caratteri di conversione d, i, n, o, u ed x possono essere preceduti da h, se l'argomento è un puntatore ad uno short invece che ad un int, oppure da l (lettera elle) se l'argomento è un puntatore a long. I ca-ratteri di conversione e, f e g possono essere preceduti da l se l'argomento è un puntatore a double e non a float, e da L se è un puntatore a long double.

int scanf(const char *format, ...)
    scanf(....) equivale a fscanf(stdin, ...)

int sscanf(char *s, const char *format, ...)
    sscanf(s, ...) equivale a scanf(....), ad eccezione del fatto che l'input viene prelevato dalla stringa s.

B1.4 Funzioni di Input / Output di Caratteri

int fgetc(FILE *stream)
    fgetc fornisce, come unsigned char (convertito ad int), il successivo carattere contenuto in stream, oppure EOF se incontra la fine del file.

char *fgets(char *s, int n, FILE *stream)
    fgets legge al più i successivi n-1 caratteri e li inserisce nel vettore s, bloccandosi prima se incon-tra un new line; il new line viene incluso nel vettore, che viene terminato con il carattere '\0'. fgets restituisce s, oppure NULL se incontra la fine del file o se rileva un errore.

int fputc(int c, FILE *stream)
    fputc scrive il carattere c (convertito ad un unsigned char) su stream. Essa restituisce il caratte-re scritto, oppure EOF in caso di errore.

int fputs(const char *s, FILE *stream)
    fputs scrive la stringa s (che può non contenere '\n') su stream; essa restituisce un valore non negativo, oppure EOF se si verifica un errore.

int getc(FILE *stream)
    getc equivale a fgetc, ad eccezione del fatto che, se è una macro, essa può valutare stream più di una volta.

`int getchar(void)`

`getchar()` equivale a `getc(stdin)`.

`char *gets(char *s)`

`gets` legge la successiva linea di input e la registra in `s`; essa sostituisce il carattere `'\n'` con `\'0'`. `gets` restituisce `s`, oppure `NULL` se raggiunge la fine del file o rileva un errore.

`int putc(int c, FILE *stream)`

`putc` equivale a `fputc`, ad eccezione del fatto che, se è una macro, essa può valutare `stream` più di una volta.

`int putchar(int c)`

`putchar(c)` equivale a `putc(c, stdout)`.

`int puts(const char *s)`

`puts` scrive su `stdout` la stringa `s` ed un new line. Essa restituisce `EOF` se rileva un errore, ed un valore non negativo in caso contrario.

`int ungetc(int c, FILE *stream)`

`ungetc` ricolloca `c` (convertito ad un `unsigned char`) su `stream`, dal quale verrà ripreso con una lettura successiva. Viene garantita la ricollocazione di un solo carattere per ogni stream. `EOF` non può essere ricollocato sul file. `ungetc` restituisce il carattere `c`, oppure `EOF` in caso di errore.

B1.5 Funzioni di Input / Output Diretto

`size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)`

`fread` legge da `stream`, collocandoli nel vettore `ptr`, al più `nobj` oggetti di ampiezza pari a `size`. `fread` restituisce il numero di oggetti letti, che può essere inferiore a quello richiesto. Per determi-nare lo stato di terminazione devono essere utilizzate le funzioni `feof` e `ferror`.

`size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)`

`fwrite` scrive su `stream`, prelevandoli dal vettore `ptr`, `nobj` oggetti di ampiezza pari a `size`. Es-sa restituisce il numero di oggetti scritti, che è inferiore a quello richiesto soltanto in caso di errore.

B1.6 Funzioni di Posizionamento su File

`int fseek(FILE *stream, long offset, int origin)`

`fseek` stabilisce la posizione sul file associato a `stream`; una successiva lettura o scrittura accede ai dati che iniziano alla nuova posizione. Per un file binario, la nuova posizione si trova a `offset` caratteri da `origin`, che può valere `SEEK_SET` (inizio del file), `SEEK_CUR` (posizione corrente) op-pure `SEEK_END` (fine del file). Per un file di testo, `offset` dev'essere zero, oppure un valore ritorna-to da `ftell` (nel qual caso `origin` deve valere `SEEK_SET`). In caso di errore `fseek` restituisce va-lori non nulli.

`long ftell(FILE *stream)`

`ftell` restituisce la posizione corrente sul file associato a `stream`, oppure, in caso di errore, `-1L`.

`void rewind(FILE *stream)`

`rewind(fp)` equivale a `fseek(fp, 0L, SEEK_SET); clearerr(fp)`.

`int fgetpos(FILE *stream, fpos_t *ptr)`

`fgetpos` memorizza in `*ptr` la posizione corrente su `stream`, per usi successivi. Il tipo `fpos_t` è quello più adatto per la registrazione di un simile valore. In caso di errore `fgetpos` restituisce un valore diverso da zero.

`int fsetpos(FILE *stream, const fpos_t *ptr)`

`fsetpos` posiziona `stream` nel punto memorizzato in `*ptr`. In caso di errore essa restituisce un valore non nullo.

B1.7 Funzioni di Errore

Molte funzioni della libreria, quando rilevano un errore, aggiornano degli indicatori di stato. Questi indicatori possono essere modificati e controllati esplicitamente. Inoltre, l'espressione intera `errno` (contenuta nello header `<errno.h>`) può contenere un numero di errore che fornisce ulteriori informazioni sull'ultimo errore verificatosi.

`void clearerr(FILE *stream)`

`clearerr` azzera gli indicatori di fine file e di errore per il file associato a `stream`.

`int feof(FILE *stream)`

`feof` restituisce un valore diverso da zero se trova ad 1 l'indicatore di fine file per `stream`.

`int ferror(FILE *stream)`

`ferror` restituisce un valore diverso da zero se trova ad 1 l'indicatore di errore per `stream`.

`void perror(const char *s)`

`perror(s)` stampa `s` ed un messaggio di errore definito dall'implementazione e corrispondente allo intero contenuto in `errno`, come se venisse impartita l'istruzione

`fprintf(stderr, "%s: %s\n", s, "`*`messaggio di errore`*`")`

Si veda, a questo proposito, la funzione `strerror` nella Sezione B3.

B2. Controlli sulla Classe dei Caratteri: CTYPE.H

L'header `<ctype.h>` dichiara le funzioni per il controllo dei caratteri. L'argomento di ognuna di queste fun-zioni è un `int`, il cui valore dev'essere `EOF` o, comunque, una quantità rappresentabile come `unsigned char`; ogni funzione ritorna un `int`. Le funzioni ritornano un valore diverso da zero (true) se l'argomento `c` soddisfa la condizione descritta, zero altrimenti.

`isalnum(c)`	`isalpha(c)` oppure `isdigit(c)` è vero
`isalpha(c)`	`isupper(c)` oppure `islower(c)` è vero
`iscntrl(c)`	carattere di controllo
`isdigit(c)`	cifra decimale
`isgraph(c)`	carattere stampabile diverso dallo spazio
`islower(c)`	lettera minuscola
`isprint(c)`	carattere stampabile compreso lo spazio
`ispunct(c)`	carattere stampabile diverso da spazio, lettera e cifra
`isspace(c)`	spazio, salto pagina, new line, return, tab, tab verticale
`isupper(c)`	lettera maiuscola
`isxdigit(c)`	cifra esadecimale

Nel set di caratteri ASCII a 7 bit, i caratteri stampabili vanno da `0x20` (`' '`) a `0x7E` (`'-'`); i caratteri di con-trollo vanno da `0` (NUL) a `0x1F` (US), più il carattere `0x7F` (DEL).

Oltre a quelle già elencate, esistono due funzioni che convertono le lettere:

`int tolower(int c)`	converte `c` in una lettera minuscola;
`int toupper(int c)`	converte `c` in una lettera maiuscola.

Se `c` è una lettera maiuscola, `tolower(c)` restituisce la corrispondente lettera minuscola, altrimenti ritorna `c`. Se `c` è una lettera minuscola, `toupper(c)` restituisce una lettera maiuscola, altrimenti restituisce `c`.

B3. Funzioni sulle Stringhe: STRING.H

Esistono due gruppi di funzioni sulle stringhe, entrambi definiti in `<string.h>`. Il primo gruppo è composto da nomi che iniziano con `str`, il secondo da nomi che iniziano con `mem`. Ad esclusione di `memmove`, il com-portamento è indefinito se si effettuano delle copie su aree che si sovrappongono.

Nella tabella seguente, le variabili `s` e `t` sono di tipo `char *`; `cs` e `ct` sono di tipo `const char *`; `n` è di tipo `size_t` e `c` è un `int` convertito ad un `char`.

`char *strcpy(s, ct)`	Copia `ct` in `s`, compreso il carattere `'\0'`; restituisce `s`.
`char *strncpy(s, ct, n)`	Copia al più `n` caratteri della stringa `ct` in `s`; restituisce `s`. Se `ct` ha meno di `n` caratteri, `s` viene riempito con una serie di `'\0'`.
`Char *strcat(s, ct)`	Concatena la stringa `ct` al termine di `s`; restituisce `s`.
`char *strncat(s, ct, n)`	Concatena al più `n` caratteri di `ct` al termine di `s`, che viene chiusa con `'\0'`; restituisce `s`.
`int strcmp(cs, ct)`	Confronta `cs` e `ct`; il valore di ritorno è <0 se `cs<ct`, 0 se `cs==ct`, >0 se `cs>ct`.
`int strncmp(cs, ct, n)`	Confronta al più `n` caratteri di `cs` con la stringa `ct`; il valore di ritorno è <0 se `cs<ct`, 0 se `cs==ct`, >0 se `cs>ct`.
`char *strchr(cs, c)`	Restituisce un puntatore alla prima occorrenza di `c` in `cs` oppure `NULL`, se `c` non compare in `cs`.
`char *strrchar(cs, c)`	Restituisce un puntatore all'ultima occorrenza di `c` in `cs`, oppure `NULL` se `c` non compare in `cs`.
`size_t strspn(cs, ct)`	Restituisce la lunghezza del prefisso di `cs` composto da caratteri di `ct`.
`size_t strcspn(cs, ct)`	Restituisce la lunghezza del prefisso di `cs` composto da caratteri che non sono di `ct`.
`char *strpbrk(cs, ct)`	Restituisce un puntatore alla prima occorrenza, in `cs`, di un qualsiasi carattere di `ct`, oppure `NULL` se `ct` non compare in `cs`.
`char *strstr(cs, ct)`	Restituisce un puntatore alla prima occorrenza della stringa `ct` in `cs`, oppure `NULL` se `ct` non compare in `cs`.
`size_t strlen(cs)`	Restituisce la lunghezza di `cs`.
`char *strerror(n)`	Restituisce un puntatore ad una stringa definita dall'implementazione ed associata all'errore `n`.
`char *strtok(s, ct)`	`strtok` cerca in `s` dei token delimitati da caratteri di `ct`. Per maggiori dettagli, si veda la spiegazione seguente.

Una sequenza di chiamate a `strtok(s, ct)` spezza `s` in una serie di token, ognuno delimitato da un ca-rattere preso da `ct`. La prima chiamata della sequenza ha il parametro `s` diverso da NULL. Esso trova il pri-mo token di `s`, composto dai caratteri non presenti in `ct`; questa chiamata termina sovrapponendo al suc-cessivo carattere di `s` il carattere `'\0'`, e restituendo un puntatore al token. Ogni chiamata successiva, indicata dal fatto che il parametro `s` è `NULL`, restituisce il successivo token costruito in questo modo, ini-ziando la scansione al termine del token precedente. `Strtok` restituisce `NULL` quando non trova più alcun token. La stringa `ct` può variare ad ogni chiamata.

Le funzioni `mem...` sono dedicate alla manipolazione di oggetti quali i vettori di caratteri; il loro scopo è quello di fornire un'interfaccia verso routine efficienti. Nella tabella seguente, `s` e `t` sono di tipo `void *`; `cs` e `ct` sono di tipo `const void *`; `n` è di tipo `size_t` e `c` è un `int` convertito ad un `unsigned char`.

`void *memcpy(s, ct, n)`	Copia `n` caratteri di `ct` in `s` e ritorna `s`.
`void *memmove(s, ct, n)`	Uguale a `memcpy`, ma funziona anche se gli oggetti si sovrappongono.
`Int memcmp(cs, ct, n)`	Confronta i primi `n` caratteri di `cs` con quelli di `ct`; il valore di ritorno coincide con quello di `strcmp`.
`Void *memchr(cs, c, n)`	Ritorna un puntatore alla prima occorrenza del carattere `c` in `cs`, op-pure `NULL` se `c` non compare nei primi `n` caratteri di `cs`.
`void *memset(s, c, n)`	Mette il carattere `c` nei primi `n` caratteri di `s`; ritorna `s`.

B4. Funzioni Matematiche: MATH.H

L'header `<math.h>` dichiara le funzioni e le macro matematiche.

Le macro `EDOM` e `ERANGE` (che si trovano in `<errno.h>`) sono valori interi costanti non nulli, usati per segnalare errori sul dominio e sul campo di esistenza delle funzioni; `HUGE_VAL` è un valore `double` positi-vo. Quando un argomento cade al di fuori del dominio di definizione di una funzione, si verifica un *errore di dominio*. In caso di errore di dominio, `errno` vale `EDOM`; il valore di ritorno dipende dall'implementazione. Quando invece il risultato della funzione non può essere rappresentato come `double`, si verifica un *errore di esistenza*. Se il risultato è troppo grande, la funzione ritorna `HUGE_VAL` con il segno corretto, e lascia in `errno` il valore `ERANGE`. Se il risultato è troppo piccolo, la funzione ritorna zero; il fatto che `errno` valga `ERANGE` dipende dall'implementazione.

Nella tabella seguente, `x` e `y` sono di tipo `double`, `n` è di tipo `int`, e tutte le funzioni restituiscono un `dou-ble`. Gli angoli, nelle funzioni trigonometriche, sono espressi in radianti.

`sin(x)`	Seno di *x*.
`cos(x)`	Coseno di *x*.
`tan(x)`	Tangente di *x*.
`asin(x)`	`sin^-1`(*x*) nel range [-π/2, π/2], *x* ∈ [-1, 1].
`Acos(x)`	`cos^-1`(*x*) nel range [0, π], *x* ∈ [-1, 1].
`Atan(x)`	`tan^-1`(*x*) nel range [-π/2, π/2].
`Atan2(y,x)`	`tan^-1`(*y/x*) nel range [-π, π].
`sinh(x)`	Seno iperbolico di *x*.
`cosh(x)`	Coseno iperbolico di *x*.
`tanh(x)`	Tangente iperbolica di *x*.
`exp(x)`	Funzione esponenziale *e^x.*
`log(x)`	Logaritmo naturale `ln(x)`, *x*>0.
`Log10(x)`	Logaritmo in base 10, `log10(x)`, *x*>0.
`Pow(x,y)`	*x^y*. Se x=0 e y<=0, oppure x<0 e y non è un intero si ha un errore di dominio.
`sqrt(x)`	Radice quadrata di *x*, *x*>=0.
`ceil(x)`	Il minimo intero non minore di *x*, espresso come `double`.
`floor(x)`	Il massimo intero non maggiore di *x*, espresso come `double`.
`fabs(x)`	Valore assoluto di *x*.
`ldexp(x,n)`	$x*2^n$.
`frexp(x, int *exp)`	Separa `x` in una frazione normalizzata nell'intervallo [-1/2, 1], che viene ritornata, ed in una potenza di 2, che viene memorizzata in `*exp`. Se `x` è ze-ro, entrambe le parti del risultato sono nulle.
`modf(x, double *ip)`	Separa `x` in una parte intera ed una frazionaria, ognuna avente il segno di `x`. La parte intera viene memorizzata in `*ip`, la parte frazionaria viene ritornata.
`fmod(x,y)`	Resto, in floating point, della divisione `x/y`, con segno uguale a quello di `x`. Se `y` è zero, il risultato dipende dall'implementazione.

B5. Funzioni di Utilità: STDLIB.H

L'header `<stdlib.h>` dichiara funzioni relative alla conversione dei numeri, all'allocazione di memoria e funzionalità simili.

`double atof(const char *s)`

`atof` converte `s` in un `double`; equivale a `strtod(s, (char**)NULL)`.

`int atoi(const char *s)`

Converte `s` in un `int`; equivale a `(int)strtol(s, (char**)NULL, 10)`.

`long atol(const char *s)`

Converte `s` in un `long`; equivale a `strtol(s, (char**)NULL, 10)`.

`double strtod(const char *s, char **endp)`

`strtod` converte il prefisso `s` in un `double`, ignorando eventuali spazi bianchi iniziali; se `endp` non è nullo, essa memorizza in `*endp` un puntatore al suffisso non convertito. Se il valore di ritorno è troppo grande, il valore restituito è `HUGE_VAL`, con il segno appropriato; se il valore di ritorno è trop-po piccolo, viene restituito uno zero. In entrambi i casi, `errno` vale `ERANGE`.

`long strtol(const char *s, char **endp, int base)`

`strtol` converte il prefisso di `s` in un `long`, ignorando eventuali spazi bianchi iniziali; se `endp` non è nullo, essa memorizza in `*endp` un puntatore al suffisso non convertito. Se `base` è compreso fra 2 e 36, la conversione viene effettuata assumendo che l'input sia scritto in quella base. Se `base` è ze-ro, la base è 8, 10 oppure 16; uno `0` iniziale indica la base ottale, ed uno `0x` (o `0X`) indica la base esadecimale. In ogni caso, le lettere rappresentano le cifre da 10 a `base-1`; in base 16 è consentito il prefisso `0x` (o anche `0X`). Se il valore di ritorno è troppo grande o troppo piccolo, il valore restituito è `LONG_MAX` o `LONG_MIN`, in base al segno del risultato, ed `errno` vale `ERANGE`.

`unsigned long strtoul(const char *s, char **endp, int base)`

`strtoul` equivale a `strtol`, con la differenza che il valore ritornato è di tipo `unsigned long` ed il valore di `errno` è `ULONG_MAX`.

`int rand(void)`

`rand` restituisce un intero pseudo-casuale compreso fra 0 e `RAND_MAX`, che vale almeno 32767.

`void srand(unsigned int seed)`

`srand` usa `seed` come seme per una nuova sequenza di numeri pseudo-casuali. Il seme iniziali è 1.

`void *calloc(size_t nobj, size_t size)`

`calloc` restituisce un puntatore allo spazio per un vettore di `nobj` oggetti di ampiezza `size`, oppu-re `NULL` se la richiesta di allocazione non può essere soddisfatta. Lo spazio è inizializzato ad una serie di zeri.

`void *malloc(size_t size)`

`malloc` restituisce un puntatore allo spazio per un oggetto di ampiezza pari a `size`, oppure `NULL` se la richiesta di allocazione non può essere soddisfatta. Lo spazio non è inizializzato.

`void *realloc(void *p, size_t size)`

`realloc` modifica, portandola a `size`, l'ampiezza dell'oggetto puntato da `p`. I contenuti restano in-variati per uno spazio pari al minimo fra la nuova e la vecchia ampiezza. Se la nuova ampiezza è maggiore della vecchia, il nuovo spazio non è inizializzato. `realloc` ritorna un puntatore alla nuova area, oppure `NULL` se la richiesta non può essere soddisfatta, nel qual caso `*p` rimane invariato.

`void free(void *p)`

`free` dealloca lo spazio puntato da `p`; essa non fa niente se `p` è NULL. `p` dev'essere un puntatore ad un'area precedentemente allocata con `calloc`, `malloc` o `realloc`.

`void abort(void)`

`abort` provoca una terminazione anomala del programma, come se venisse impartita l'istruzione `raise(SIGABRT)`.

`void exit(int status)`

`exit` provoca la terminazione normale del programma. Le funzioni `atexit` vengono chiamate in ordine inverso rispetto alla registrazione, i buffer associati a file aperti vengono scaricati, gli stream aperti vengono chiusi ed il controllo viene restituito all'ambiente chiamante. Il modo in cui lo stato viene restituito all'ambiente esterno dipende dall'implementazione, anche se lo zero indica sempre una terminazione corretta. Possono essere utilizzati anche i valori `EXIT_SUCCESS` e `EXIT_FAI-LURE`.

`int atexit(void (*fcn)(void))`

`atexit` memorizza il fatto che la funzione `fcn` dev'essere chiamata al momento della terminazione normale del programma; essa restituisce un valore non nullo se la registrazione non può essere ef-fettuata.

`int system(const char *s)`

`system` passa all'ambiente esterno la stringa `s` affinché venga eseguita. Se `s` è `NULL`, `system` ritor-na un valore non nullo se esiste un command processor. Se `s` è diversa da `NULL`, il valore di ritorno dipende dall'implementazione.

```
char *getenv(const char *name)
```
getenv restituisce la stringa dell'ambiente associato a name, oppure NULL se tale stringa non esi-ste. I dettagli dipendono dall'implementazione.

```
void *bsearch(const void *key, const void *base, size_t n, size_t size,
              int (*cmp)(const void *keyval, const void *datum))
```

bsearch cerca in base[0]...base[n-1] un elemento uguale a quello in *key. La funzione cmp deve ritornare un valore negativo se il suo primo argomento (la chiave di ricerca) è inferiore al secondo (un ele-mento della tabella), zero se sono uguali, ed un valore positivo se il primo argomento è maggiore del secon-do. Gli elementi nel vettore base devono essere ordinati in ordine crescente. bsearch ritorna un puntatore all'elemento trovato, o NULL se esso non esiste.

```
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *, const void *))
```
qsort ordina in ordine crescente un vettore base[0]...base[n-1] di oggetti di ampiezza pari a size. La funzione di confronto cmp deve avere caratteristiche analoghe a quelle descritte per bsearch.

```
int abs(int n)
```
abs restituisce il valore assoluto del suo argomento (un int).

```
long labs(long n)
```
labs restituisce il valore assoluto del suo argomento (un long).

```
div_t div(int num, int denom)
```
div calcola il quoziente ed il resto della divisione num/denom. I risultati vengono memorizzati nei membri, di tipo int, quot e rem di una struttura di tipo div_t.

```
ldiv_t ldiv(long num, long denom)
```
ldiv calcola il quoziente ed il resto della divisione num/denom. I risultati vengono memorizzati nei membri, di tipo long, quot e rem di una struttura di tipo ldiv_t.

B6. Diagnostica: ASSERT.H

La macro assert viene utilizzata per aggiungere messaggi diagnostici ai programmi:

```
void assert(int espressione)
```

Se *espressione* è zero quando viene eseguita

```
assert(espressione)
```

la macro assert stampa su stderr un messaggio, come per esempio

```
Assertion failed: espressione, file nomefile, linea nnn
```

Dopo la stampa, per terminare l'esecuzione assert chiama abort. Il nome del file sorgente ed il numero di linea provengono dalle macro di preprocessor __FILE__ e __LINE__.

Se quando viene incluso <assert.h> la variabile NDEBUG è definita, la macro assert viene ignorata.

B7. Liste Variabili di Argomenti: STDARG.H

L'header <stdarg.h> contiene funzioni che consentono di scandire una lista di argomenti variabili in nu-mero e tipo.

Supponiamo che *lastarg* sia l'ultimo parametro esplicito di una funzione `f`, avente un numero variabile di ar-gomenti. Dichiariamo allora, in `f`, una variabile `ap` di tipo `va_list`, che punta di volta in volta ad uno dei diversi argomenti:

```
va_list ap;
```

`ap` deve venire inizializzato una volta, tramite la macro `va_start`, prima che uno qualsiasi degli argomenti non elencati venga utilizzato:

```
va_start(va_list ap, lastarg);
```

Da questo momento, ogni esecuzione della macro `va_arg` produrrà un valore avente il tipo ed il valore del successivo argomento non specificato, e modificherà `ap` in modo che la chiamata seguente a `va_arg` resti-tuisca l'argomento successivo:

```
type va_arg(va_list ap, type);
```

Dopo avere trattato tutti gli argomenti e prima di uscire da `f`, è necessario invocare una volta la macro

```
void va_end(va_list arg);
```

B8. Salti non Locali: SETJMP.H

Le dichiarazioni contenute in <setjmp.h> permettono di alterare l'esecuzione della normale sequenza di chiamata ed uscita da una funzione, tipicamente per consentire un ritorno immediato da una chiamata di funzione profondamente annidata.

`int setjmp(jmp_buf env)`

La macro `setjmp` memorizza in `env` le informazioni sullo stato necessarie per l'esecuzione di `longjmp`. Il valore di ritorno è zero in caso di chiamata diretta a `setjmp`, diverso da zero in caso di una chiamata conseguente a `longjmp`. Una chiamata a `setjmp` può comparire soltanto in alcuni contesti: le parti di controllo dei costrutti `if` e `switch`, i cicli ed alcune semplici espressioni relazio-nali.

```
if (setjmp(env)==0)
    /* codice eseguito su chiamata diretta */
else
    /* codice eseguito su chiamata a longjmp */
```

`void longjmp(jmp_buf env, int val)`

`longjmp` ripristina lo stato salvato dall'ultima chiamata a `setjmp`, usando le informazioni contenu-te in `env`, e l'esecuzione riprende come se la funzione `setjmp` fosse stata eseguita ed avesse resti-tuito il valore `val`, diverso da zero. La funzione che contiene la chiamata a `setjmp` deve non esse-re ancora terminata. Gli oggetti accessibili hanno i valori che possedevano al momento della chia-mata a `longjmp`; questi valori non vengono salvati da `setjmp`.

B9. Segnali: SIGNAL.H

L'header <signal.h> fornisce funzioni per la gestione di condizioni di eccezione che si verificano durante l'esecuzione, come l'arrivo di un segnale di interrupt da una sorgente esterna, oppure un errore nell'esecu-zione.

`void (*signal(int sig, void (*handler)(int)))(int)`

`signal` determina il modo in cui i successivi segnali verranno gestiti. Se `handler` vale `SIG_DFL`, viene adottato il comportamento di default, dipendente dall'implementazione; se `handler` vale, in-vece, `SIG_DFL`, il segnale viene ignorato; altrimenti, all'atto della ricezione del segnale viene invo-cata la funzione puntata da `handler`. I segnali validi comprendono

`SIGABRT`	Terminazione anomala, per esempio dovuta ad una `abort`.
`SIGFPE`	Errore aritmetico, per esempio divisione per zero o overflow.
`SIGILL`	Immagine illegale della funzione, per esempio istruzione illegale.

SIGINT	Segnalazione interattiva, per esempio interrupt.
SIGSEGV	Illegale accesso a memoria, per esempio riferimento ad un indirizzo esterno alla memoria.
SIGTERM	Richiesta di terminazione inviata a questo programma.

`signal` restituisce il precedente valore di `handler` per uno specifico segnale, oppure `SIG_ERR` se rileva un errore.

Quando il segnale `sig` viene effettivamente ricevuto, il segnale viene riportato al suo comportamento di de-fault; quindi, viene eseguita la funzione di gestione del segnale, come se venisse impartita l'istruzione `(*handler)(sig)`. Se tale funzione ha un punto di ritorno, l'esecuzione riprende dal punto in cui era stato ricevuto il segnale.

Lo stato iniziale dei segnali dipende dall'implementazione.

`int raise(int sig)`

`raise` invia il segnale `sig` al programma; se si verifica un errore, essa ritorna un valore non nullo.

B10. Funzioni Relative a Data e Ora: TIME.H

L'header `<time.h>` dichiara tipi e funzioni che consentono di manipolare l'ora e la data. Alcune funzioni gestiscono l'ora locale, che può variare, per esempio, al variare della zona oraria. `clock_t` e `time_t` sono tipi aritmetici che rappresentano il tempo, e `struct tm` contiene le componenti di un calendario:

`int tm_sec;`	Secondi dopo il minuto (0, 61).
`int tm_min;`	Minuti dopo l'ora (0, 59).
`int tm_hour;`	Ore dopo mezzanotte(0, 23).
`int tm_mday;`	Giorno del mese (1, 31).
`int tm_mon;`	Mesi dopo Gennaio (0, 11).
`int tm_year;`	Anni dopo il 1900.
`int tm_wday;`	Giorni dopo la Domenica (0, 6).
`int tm_yday;`	Giorni dopo Gennaio (0, 365).
`int tm_isdst;`	Flag per l'ora legale.

`tm_isdst` è positivo se l'ora legale è in vigore, nullo se non lo è e negativo se quest'informazione non è disponibile.

`clock_t clock(void)`

`clock` restituisce il tempo di CPU usato dal programma dall'inizio dell'esecuzione, e `-1` se tale in-formazione non è disponibile. `clock()/CLOCK_PER_SEC` è un tempo espresso in secondi.

`time_t time(time_t *tp)`

`time` restituisce l'ora corrente, oppure `-1` se essa non è disponibile. Se `tp` è diverso da `NULL`, il valore di ritorno viene anche assegnato a `*tp`.

`double difftime(time_t time2, time_t time1)`

`diffitime` ritorna, esprimendola in secondi, la differenza `time2-time1`.

`time_t mktime(struct tm *tp)`

`mktime` converte l'ora locale contenuta nella struttura `*tp` nell'ora corrente, con la stessa rappre-sentazione usata da `time`. Le componenti avranno valori compresi negli intervalli sopra elencati. `mktime` restituisce l'ora, oppure `-1` se essa non può essere rappresentata.

Le prossime quattro funzioni restituiscono puntatori ad oggetti statici che possono venire sovrascritti nel corso di chiamate successive.

`char *asctime(const struct tm *tp)`

`asctime` converte la data contenuta nella struttura `*tp` in una stringa avente il seguente formato

```
Sun Jan  3  15:14:13 1988\n\0
```

`char *ctime(const time_t *tp)`
: `ctime` converte l'ora contenuta in `*tp` nell'ora locale; essa equivale a

```
asctime(localtime(tp))
```

`struct tm *gmtime(const time_t *tp)`
: `gmtime` converte l'ora in `*tp` in Ora Assoluta (UTC, "Coordinated Universal Time"). Essa restituisce `NULL` se il formato UTC non è disponibile. Il nome `gmtime` deriva da motivazioni storiche.

`struct tm *localtime(const time_t $tp)`
: `localtime` converte in ora legale l'ora contenuta in `*tp`.

`size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)`
: `strftime` formatta le informazioni contenute in `*tp` secondo le direttive che compongono `fmt` (analogo al parametro `fmt` di `printf`), e pone in `s` il risultato. I caratteri normali (compreso `'\0'`) vengono copiati in `s`. Ogni `%c` viene sostituito secondo quanto descritto nel seguito, usando i valori corretti rispetto all'ambiente locale. In `s` vengono collocati al massimo `smax` caratteri. `strftime` re-stituisce il numero di caratteri, escluso `'\0'`, oppure zero se il numero dei caratteri prodotti supera `smax`.

`%a`	Nome abbreviato del giorno della settimana.
`%A`	Nome completo del giorno della settimana.
`%b`	Nome abbreviato del mese.
`%B`	Nome completo del mese.
`%c`	Rappresentazione locale di data ed ora.
`%d`	Giorno del mese (01-31)
`%H`	Ora (orologio di 24 ore) (00-23).
`%I`	Ora (orologio di 12 ore) (01-12).
`%j`	Giorno dell'anno (001-366).
`%m`	Mese (01-12).
`%M`	Minuti (00-59).
`%p`	Equivalente locale di AM o PM.
`%S`	Secondi (00-61).
`%U`	Numero della settimana (inizia di Domenica) nell'anno (00-53).
`%w`	Giorno della settimana (0-6, Domenica è il giorno 0).
`%W`	Numero della settimana (che inizia di Lunedì) nell'anno (00-53).
`%x`	Rappresentazione locale della data.
`%X`	Rappresentazione locale dell'ora.
`%y`	Anno senza il secolo (00-99).
`%Y`	Anno con il secolo.
`%Z`	Nome della zona oraria, se c'è.
`%%`	%

B11. Limiti Definiti dall'Implementazione: LIMITS.H

L'header `<limits.h>` definisce alcune costanti di tipo intero. I valori riportati nel seguito sono le grandez-ze minime consentite; possono venire utilizzati anche valori superiori.

`CHAR_BIT`	`8`	Bit di un `char`.
`CHAR_MAX`	`UCHAR_MAX` *o* `SCHAR_MAX`	Massimo valore di un `char`.
`CHAR_MIN`	`0` *o* `SCHAR_MIN`	Minimo valore di un `char`.
`INT_MAX`	`+32767`	Massimo valore di un `int`.
`INT_MIN`	`-32767`	Minimo valore di un `int`.
`LONG_MAX`	`+2147483647`	Massimo valore di un `long`.
`LONG_MIN`	`-2147483647`	Minimo valore di un `long`.
`SCHAR_MAX`	`+127`	Massimo valore di un `signed char`.
`SCHAR_MIN`	`-127`	Minimo valore di un `signed char`.
`SHRT_MAX`	`+32767`	Massimo valore di un `short`.

SHRT_MIN	-32767	Minimo valore di un short.
UCHAR_MAX	255	Massimo valore di un unsigned char.
UINT_MAX	65535	Massimo valore di un unsigned int.
ULONG_MAX	4294967295	Massimo valore di un unsigned long.
USHRT_MAX	65535	Massimo valore di un unsigned short.

I nomi della tabella che segue, un sottoinsieme di <float.h>, sono quelli di costanti legati all'aritmetica in floating point. Quando il valore viene fornito, esso rappresenta la grandezza minima della quantità corri-spondente. Ogni implementazione definisce valori appropriati.

FLT_RADIX	2	Radice della rappresentazione esponenziale; per esempio 2, 16.
FLT_ROUNDS		Modalità di arrotondamento del floating point per la somma.
FLT_DIG	6	Cifre decimali della precisione.
FLT_EPSILON	1E-5	Numero minimo x tale che $1.0+x \neq 1.0$.
FLT_MANT_DIG		Numero di cifre in base FLT_RADIX della mantissa.
FLT_MAX	1E+37	Massimo numero floating point.
FLT_MAX_EXP		Massimo n tale che FLT_RADIX^n-1 sia rappresentabile.
FLT_MIN	1E-37	Minimo numero floating point normalizzato.
FLT_MIN_EXP		Minimo n tale che 10^n sia un numero normalizzato.
DBL_DIG	10	Cifre decimali della precisione.
DBL_EPSILON	1E-9	Minimo numero x tale che $1.0+x \neq 1.0$.
DBL_MANT_DIG		Numero di cifre in base FLT_RADIX della mantissa.
DBL_MAX	1E+37	Massimo numero double floating point.
DBL_MAX_EXP		Massimo n tale che FLT_RADIX^n-1 sia rappresentabile.
DBL_MIN	1E-37	Minimo numero double floating point normalizzato.
DBL_MIN_EXP		Minimo n tale che 10^n sia un numero normalizzato.

APPENDICE C

SOMMARIO DELLE VARIAZIONI

Dal momento della pubblicazione della prima edizione di questo libro, la definizione del linguaggio ha subito delle modifiche. Quasi tutte queste modifiche erano delle estensioni del linguaggio, attentamente progettate in modo da restare compatibili con le regole già esistenti; alcune modifiche, invece, eliminavano delle ambiguità presenti nella descrizione originale; altre, infine, rappresentano alterazioni delle regole preesistenti. Molte delle nuove funzionalità sono state annunciate nei documenti che accompagnavano i compilatori forniti dalla AT&T, e sono state successivamente adottate anche dalle altre ditte produttrici di compilatori. Recentemente il comitato ANSI, standardizzando il linguaggio, ha incorporato la massima parte di queste modifiche, e ne ha introdotte altre. Il documento finale del comitato è stato in parte anticipato da alcuni compilatori commerciali, prodotti prima che fosse effettivamente disponibile una versione formale definitiva del linguaggio C standard.

Questa Appendice riassume le differenze tra il linguaggio definito nella prima edizione di questo libro e quello che costituirà lo Standard finale. Essa si riferisce unicamente al linguaggio, e tralascia l'ambiente in cui esso opera e la libreria; pur essendo anche questi aspetti importanti dello Standard, un confronto con la prima edizione sarebbe pressoché impossibile, perché essa non li trattava.

- Nello Standard il preprocessing è definito più attentamente rispetto alla prima edizione, ed è più esteso: esso si basa esplicitamente sui token; possiede nuovi operatori per la concatenazione dei token (`##`), e per la creazione di stringhe (`#`); prevede nuove linee di controllo come `#elif` e `#pragma`; è esplicitamente consentita la ridichiarazione di macro che utilizzino la stessa sequenza di token; i parametri all'interno delle stringhe non vengono più sostituiti. L'unione di due linee tramite `\` è consentita ovunque, non più soltanto nella definizione delle macro e nelle stringhe. Si veda il paragrafo A12.

- Il numero minimo di caratteri significativi per gli identificatori interni è stato portato a 31; quello per gli identificatori esterni rimane, invece, di 6 caratteri tutti maiuscoli o tutti minuscoli (molte implementazioni hanno aumentato questo minimo).

- L'introduzione delle sequenze triplici, inizianti con `??`, consente di rappresentare caratteri che non sono presenti in alcuni set. Sono state definite sequenze `#\^[]{}#~`; si veda il paragrafo A12.1. Osservate che l'introduzione delle sequenze triplici può modificare il significato di stringhe contenenti la sequenza `??`.

- Sono state introdotte nuove parole chiave (`void`, `const`, `volatile`, `signed`, `enum`). È stata eliminata la parola chiave `entry`, nata nella prima edizione ma mai utilizzata.

- Sono state definite nuove sequenze di escape, da utilizzare all'interno di costanti carattere di stringhe letterali. Se il carattere `\` è seguito da un carattere che non fa parte di una sequenza di escape approvata, il comportamento è indefinito. Si veda il paragrafo A2.5.2.

- Una modifica banale, desiderata da tutti: 8 e 9 non sono cifre ottali.

- Lo Standard introduce un più vasto insieme di suffissi che rendono esplicito il tipo delle costanti: `U` o `L` per gli interi, `F` o `L` per i floating. Esso perfeziona anche le regole per la determinazione del tipo delle costanti prive di suffisso (paragrafo A2.5).

- Stringhe letterali adiacenti vengono concatenate.

- Esiste una notazione particolare per le stringhe letterali e le costanti composte da caratteri estesi; si veda il paragrafo A2.6.

- I caratteri, analogamente ad altri tipi, possono essere esplicitamente dichiarati come privi di segno o meno, usando rispettivamente le parole chiave `unsigned` e `signed`. È stata eliminata la locuzione `long float`, sinonimo di `double`; `long double`, però, può venire utilizzato per quantità floating in extra precisione.

- Per un certo periodo è stato disponibile il tipo `unsigned char`. Lo Standard ha introdotto la parola chiave `signed` per rendere esplicita la presenza del segno per gli oggetti `char` e per quelli di tutti gli altri tipi interi.

- Il tipo `void` era disponibile, su alcune implementazioni, già da alcuni anni. Lo Standard introduce l'uso del tipo `void *`, che identifica il puntatore generico; in precedenza, questo ruolo era coperto dal tipo `char *`. Nello stesso tempo, sono state emanate regole per impedire la commistione fra puntatori ed interi, e fra puntatori di tipo diverso, senza l'uso dell'operatore di casting.

- Lo Standard stabilisce dei valori minimi per i tipi aritmetici, e degli appositi header (`<limits.h>` e `<float.h>`) che contengono i valori specifici di ogni implementazione.

- Rispetto alla prima edizione di questo libro, le enumerazioni sono un'innovazione.

- Lo Standard adotta dal C++ la nozione di qualificatore di tipo, per esempio `const` (paragrafo A8.2).

- Le stringhe non sono più modificabili, e quindi potrebbero essere poste in memoria a sola lettura.

- Sono cambiate le "conversioni aritmetiche usuali", essenzialmente passate da "per gli interi, prevale sempre lo specificatore `unsigned`; per i floating, prevale sempre `double`" a "il tipo finale è il più piccolo tipo che può rappresentare la quantità voluta". Si veda il paragrafo A6.5.

- I vecchi operatori di assegnamento come `=+` sono scomparsi. Inoltre, ora gli operatori di assegnamento sono dei singoli token; nella prima edizione, essi erano delle coppie, e potevano venire sepa-rati con degli spazi bianchi.

- È stata tolta ai compilatori la possibilità di decidere se trattare come computazionalmente associativi gli operatori matematicamente associativi.

- Per simmetria con l'operatore unario `-`, è stato introdotto l'operatore unario `+`.

- Un puntatore ad una funzione può venire utilizzato come designatore di funzione senza che debba essere usato esplicitamente l'operatore `*`. Si veda il paragrafo A7.3.2.

- Le strutture possono essere assegnate, passate alle funzioni e ritornate dalle funzioni.

- È consentita l'applicazione dell'operatore di indirizzamento `&` ai vettori, ed il risultato è un puntatore al vettore.

- L'operatore `sizeof`, nella prima edizione, produceva il tipo `int`; successivamente, molte implementazioni trasformarono questo `int` in un `unsigned`. Lo Standard rende esplicitamente dipendente dall'implementazione questo tipo, ma richiede che esso, `size_t` venga definito nell'header `<stddef.h>`. Una modifica analoga è stata introdotta per il tipo (`ptrdiff_t`) della differenza tra due puntatori. Si vedano i paragrafi A7.4.8 e A7.7.

- L'operatore di indirizzamento `&` non può venire applicato ad un oggetto dichiarato `register`, anche se l'implementazione sceglie di non tenere tale oggetto in un registro.

- Il tipo di un'espressione di shift è quello del suo operando sinistro; l'operando destro non può determinare il tipo del risultato. Si veda il paragrafo A7.8.

- Lo Standard legalizza la creazione di un puntatore che cada subito dopo il termine di un vettore, e consente che gli vengano applicate relazioni aritmetiche; si veda il paragrafo A7.7.

- Lo Standard introduce (ereditandola dal C++) la nozione di una dichiarazione di un prototipo di funzione che incorpori il tipo dei parametri, ed include il riconoscimento esplicito di funzione con numero di argomenti variabile, insieme ad un ben definito procedimento per il loro trattamento. Si vedano i paragrafi A7.3.2, A8.6.3, B7. Il vecchio stile, anche se con delle restrizioni, è ancora accettato.

- Le dichiarazioni vuote, che non hanno dichiaratori e non dichiarano almeno una struttura, una union od un'enumerazione sono proibite nello Standard. D'altro canto, una dichiarazione con un tag di una struttura o union ridichiara quel tag, anche se esso era già stato dichiarato in uno scope più esterno.

- Sono vietate le dichiarazioni esterne di dati prive di specificatori o qualificatori (cioè i dichiaratori puri).

- Alcune implementazioni, quando trovavano una dichiarazione `extern` all'interno di un blocco, la esportavano al resto del file. Lo Standard chiarisce che lo scope di una simile dichiarazione rimane il blocco.

- Lo scope dei parametri è l'istruzione composta di una funzione, in modo che le dichiarazioni delle variabili all'inizio della funzione non possano nascondere i parametri.

- Sono in parte cambiati gli spazi dei nomi degli identificatori. Lo Standard colloca tutti i tag in un singolo spazio dei nomi, ed introduce uno spazio separato per le label; si veda il paragrafo A11.1. Inoltre, i nomi dei membri sono associati alla struttura o union alla quale appartengono (nella pratica, questa convenzione è stata adottata già da qualche tempo).

- Le union possono venire inizializzate; l'inizializzatore si riferisce al primo membro.

- Le strutture, le union ed i vettori automatici possono venire inizializzati, anche se con delle restrizioni.

- I vettori di caratteri aventi una dimensione esplicita possono venire inizializzati con una stringa letterale composta da un numero di caratteri pari a tale dimensione (il carattere `\0` non viene considera-to).

- L'espressione di controllo, e le label dei vari casi, di uno `switch` possono essere di qualsiasi tipo intero.